PERT
A PIPELINED ENGINE FOR RAY TRACING GRAPHICS

by

Pradeep Chilka

B.Tech., Banaras Hindu University, 1979

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the Department
of
Computing Science

SIMON FRASER UNIVERSITY

August 1985

Approval

Name: Pradeep Chilka

Degree: Master of Science

Title of Thesis: PERT: A Pipelined Engine for Ray Tracing Graphics

Examining Committee:

Chairperson: Dr. Art Liestman

Dr. Richard Hobson
Senior Supervisor

Dr. Thomas Calvert

Dr. ~~Louis Hafer~~

Roy Hall
Graphics Consultant,
External Examiner
(in absentia)

24 May 1985

Date Approved:

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

PERT: A Pipelined Engine for Ray Tracing Graphics

Author:

(signature)

Pradeep Chilka

(name)

1985 August 14

(date)

Abstract

Ray tracing techniques for image rendering have produced some of the most realistic images to date. Ray tracing, however, is computationally expensive because of the floating point calculation involved in ray-object intersection and the number of such intersections that must be performed to render an image realistically Conventional mini-computers take anywhere between an hour and several days to render a single image of moderate complexity.

In this thesis, we propose a pipelined machine, PERT, which according to our simulation results, shows a substantial reduction in the rendering time.

The key features of PERT are: a) the use of bounding volumes and hierarchical data organization to reduce the number of ray-object intersections, b) a 3-processor pipeline that executes a 3-task ray tracing algorithm, c) microcoded, custom designed, VLSI processors in each stage of the pipeline, and d) extensibility to a multi-PERT architecture that consists of several PERTs working in parallel.

To my parents

"in the beginning was the Word "

John, *1.1*

"in the beginning was the Word all right, but it wasn't a fixed number of bits"

R.S. Barton, *Software Engineering*

Acknowledgements

I wish to thank the following people:

Dr. Rick Hobson, my senior supervisor, for his patience and guidance throughout the course of this work.

Dr. Tom Calvert, Dr. Lou Hafer, and Dr. Binay Bhattacharya for their many thoughtful contributions to my work.

Severin Gaudet, my research partner, with whom this thesis was carried out as a joint project.

This work has been supported by the Science Council of B.C. grant #40 (RC-10)

Table of Contents

List of Figures

List of Tables

Chapter 1

RAY TRACING

The potential of ray tracing techniques to produce realistic images has been extolled by so many that it is on the verge of becoming a cliche. Nevertheless, the images speak for themselves; images which can be virtually indistinguishable from photographs. These realistic images are a product of both good scene descriptions or models which describe the shape and position of objects, and good rendering techniques. We are concerned with the latter In this chapter we shall discuss what creates the illusion of realism and why ray tracing techniques are capable of exploiting this.

1.1. The Shading Model

As stated above, ray tracing techniques have generated some of the most realistic images to date. To understand what contributes to the realism of a synthetic image, one must first understand the process that occurs naturally in the real world.

It is generally accepted that a colour video camera produces a realistic image. So let us first consider how the camera records a scene onto the phosphors or pixels of a monitor. Imagine that for each pixel on the monitor screen, these is a corresponding sensor on the camera s focal plane behind the lens. The surfaces in the scene visible to the camera reflect or transmit light into the lens and onto the sensors that in turn measure the light and send signals to their respective display pixels. The colour of each pixel is determined by the colour of the corresponding area in the scene. The colour of a surface is

determined by the properties of the surface and the light falling on the it; this means we have to know how the light interacts with the surface.

In rendering a scene, it must be possible to model these light interactions in order to simulate the light being reflected or transmitted to the sensors. Examination of the light falling upon an area of the surface allows it to be classified in one of two ways. The first is light coming directly from emitting sources (eg. the sun, an incandescent bulb, a flourescent tube); this type of light is referred to as a direct source. The second type is light being reflected onto the surface from other surfaces; this constitutes an indirect or global source.

Next examining the surface with which these two sources of light interact, we can distinguish three surface characteristics which influence these interactions. The first of these is the roughness of the surface at the microscopic level. This determines how light falling on the surface is scattered by reflection in all directions and thus how good it is as a diffuse reflector. The second characteristic, the opposite of the first, is the smoothness at the microscopic level that in turn determines the degree to which the surface can be characterized as a mirror; this property results in a specular reflection. Finally the third characteristic determines how well a surface transmits light from a light source from behind.

Combining these characteristics with the types of light sources, a formula can be derived which models the cumulative effect of the six combinations according to the physical laws of optics. This formula is referred to as the shading or illumination model. When rendering an image, we can now model the interaction of light with a surface by applying the shading model to the point being examined. Consequently, it is the completeness of

the shading model which determines the degree of realism of a computer generated image. Figure 1-1 shows examples of the same scene with different light interactions being modeled.

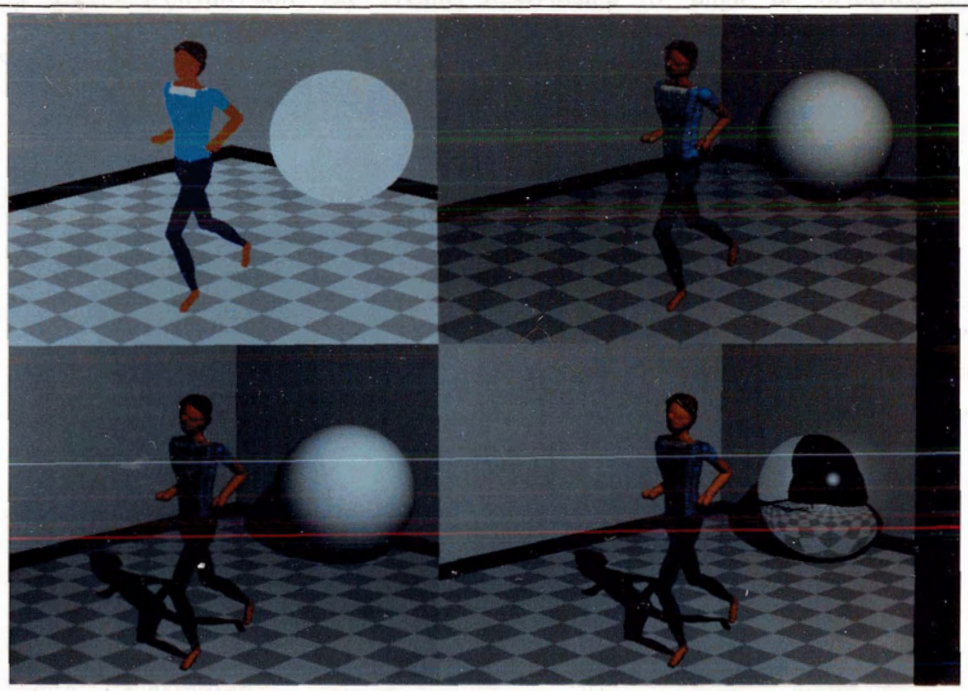

Figure 1-1: Examples of different light interactions

Shading models have become more sophisticated since the early days of computer graphics when diffuse Lambertian shading (direct source diffuse reflection) was used. In a sense, the evolution in the shading model can be compared to the evolution in painting that occurred with the Italian renaissance when the flat two dimensional-like Byzantine technique was surpassed by the vibrant realism of Michelangelo and Raphael with their studies of both light and form.

This evolution toward a better shading model began when Phong [PHON73] proposed a shading model based on empirical observations which included a term for direct source

specular reflection and global source diffuse reflection. Blinn [BLIN77], Kay [KAY79], Whitted [WHIT80], and Cook and Torrance [COOK82] have contributed to making shading models more physical and less empirical by defining terms for, among other things, global and direct source transmission, the Fresnel relationship for angle of incidence, and direct source specular reflection. Most of these contributions have been brought together nicely by Hall [HALL83] in his shading model that is illustrated in Figure 1-2.

$$I = k_d \sum_{j=1} (\bar{N}\ \bar{L}_j) R_d I_j$$
direct diffuse

$$+ k_s \sum_{j=1} (\bar{N}\ \bar{H}_j)^n R_f I_j$$
direct reflected

$$+ k_s \sum_{j=1} (\bar{N}\ \bar{H}_j)^n T_f I_j$$
direct transmitted

$$+ k_s R_f I_r F_r^{dr}$$
global reflected

$$+ k_s T_f I_t F_t^{dt}$$
global transmitted

$$+ I_a R_d$$
global diffuse

dr = distance of reflected ray travel
dt = distance of refracted ray travel
F_r = trans per unit length of reflected ray
F_t = trans per unit length of refracted ray
$\bar{H}$ = unit reflection mirror-direction vector
$\bar{H}$ = unit trans. mirror direction vector
I = intensity of point
I_a = intensity of global ambient light
I_j = intensity of jth direct light source
I_r = intensity of reflected ray
I_t = intensity of refracted ray
j = direct light source index
k_d = diffuse reflection coefficient
k_s = specular reflection coefficient
l = number of direct light sources
$\bar{L}$ = unit light source vector
n = exponent for glossiness
$\bar{N}$ = unit surface normal vector
R_f = Fresnel reflectance curve
R_d = diffuse reflectance curve
T_f = Fresnel transmission curve

Figure 1-2: The Hall shading model

1.2. Ray Tracing

When rendering an image from a 3-dimensional scene model, the following two functions are executed: a) the visibility of the surfaces is determined with respect to the viewpoint and b) light interaction with the visible surfaces and the production of colour is characterized. Most rendering techniques, such as z-buffering, cannot exploit the complex shading models because they determine visibility by projecting the 3-D modeling space onto the 2-D image plane and thus lose the third dimension necessary for the simulation of the light interactions

Ray tracing, on the other hand can exploit the shading models because it determines visibility not on the 2-D image plane but in the 3-D modeling space. The origin of ray tracing is found in ray casting that was proposed by Appel [APPE68] and implemented by Goldstein and Nagle at MAGI [GOLD71] as a visible surface algorithm. However, Whitted's classic algorithm [WHIT80] brought ray casting and a good shading model together in the technique now known as ray tracing.

Going back to the example of the colour video camera, ray tracing simulates its operation in reverse Instead of than recording the light rays being reflected from the visible surfaces through the lens and onto the sensors, ray tracing sends out rays originating at each sensor on the focal plane (*image plane*) through the lens (*focal point*) into the scene (a model described in 3-D). An *initial ray* for each pixel of the image plane is sent out in this manner. Each ray is then intersected with each object in the scene to find the closest surface that is visible.

Once the nearest intersection point is found, the shading model is used to compute the colour. This involves spawning the following rays from the intersection point:

1. toward each direct light source in the scene (*light rays*) to determine if it is visible to the point and what contribution it makes to the diffuse, specular and transmitted components of the shading model;

2. in the mirror reflection direction (*reflected ray*) to determine the light intensity coming from that direction for calculation of the global source specular component; and

3. in the refracted ray direction (*transmitted ray*) to determine the light intensity from that direction for calculation of the global transmitted component of the shading model.

The algorithm's elegance lies in recursion because once spawned, the reflected and transmitted rays are traced in the same fashion as the initial rays If these rays intersect other surfaces, the shading model is applied and new rays are spawned until the rays leave the scene or intersect a non-reflecting surface. In this fashion the intersection tree for each pixel is built up The intersection tree has at its root the pixel, interior nodes are intersected surfaces and leaves are direct light sources or the exterior of the scene. The branches of the intersection tree are the rays spawned during the tracing of the pixel.

Figure 1-3 follows the tracing of a ray and the resulting intersection tree. An initial ray (ir) strikes object 1 (O1). The shading model is applied at the intersection point and three secondary rays are spawned. Light ray 2 (lr2) is blocked and thus ignored. The reflected ray (rr1) strikes the semi-transparent object 2 (O2). Again, secondary rays are spawned. The reflected ray (rr2) leaves the scene and is ignored. The transmitted ray (tr1) would be traced further.

Once all the rays have been traced for a pixel, the intersection tree will contain all the light source information in the leaf nodes and all the surface characteristic information in the interior nodes. The tree is traversed in a depth-first order to calculate the final pixel colour.

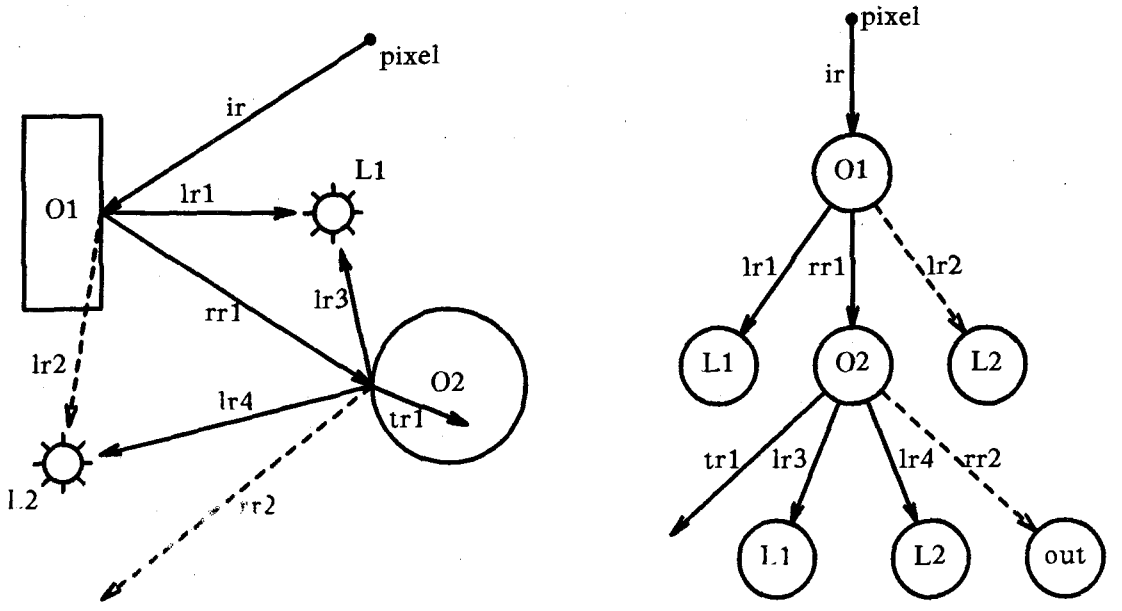

Figure 1-3: Example of the tracing of a pixel and the building of the intersection tree

1.3. An Analysis

As shown, ray tracing is a simple recursive algorithm which exploits a good shading model However, the obvious advantages of using ray tracing are almost outweighed by its principal disadvantage: computational cost. As an illustration of how severe this is, most of the reported times for published images rendered using DEC VAX/780's have been measured in hours.

Why is the algorithm so computationally intensive?

- all computations are executed in floating point.
- extensive use is made of the square root function for vector normalization of rays, normals and dot products.

- complex intersection computations are required for some classes of objects such as fractals and 3-D spline surfaces.
- the number of intersection calculations is large since determination of the closest surface requires that a ray be tested be tested against all objects in the scene.
- the number of rays spawned during the ray tracing process is also large.

To show the sheer number of computations required in ray tracing an image, we shall use an analysis of the complexity of ray tracing similar to that found in [DIPP84]. We shall also use data from the run-time profile of the program used to generate Figure 1-4 on a DEC VAX/750 with a floating point unit. To do this, we make the following assumptions:

- each intersection tree has depth $D = 4$.
- the average number of recursive reflected and transmitted rays spawned per intersection $N = 1.1$ (100% of the intersections will spawn a reflected ray; 10%, a transmitted ray).
- the number of objects in the scene $O = 1093$ (833 spheres and 260 polygons) which corresponds to the scene model used to generate Figure 1-4.
- the number of direct light sources $L = 1$.
- the resolution of the image $R_0 = 512 \times 384 = 196608$ pixels.
- the average intersection calculation time $T_i = 0.000429$ seconds.
- the average ray spawning time $T_s = 0.000710$ seconds.

The resulting calculations are given below:

- total number of rays traced: $R_t = R_0 \frac{\bullet(N^D-1)}{\bullet(N-1)} (1+L) = 1824915.$
- total number of intersections: $I_t = OR_t \sim 2^9.$

- total time: $T_t = R_t(T_s + T_i O)$ *Approx* 238 hours.

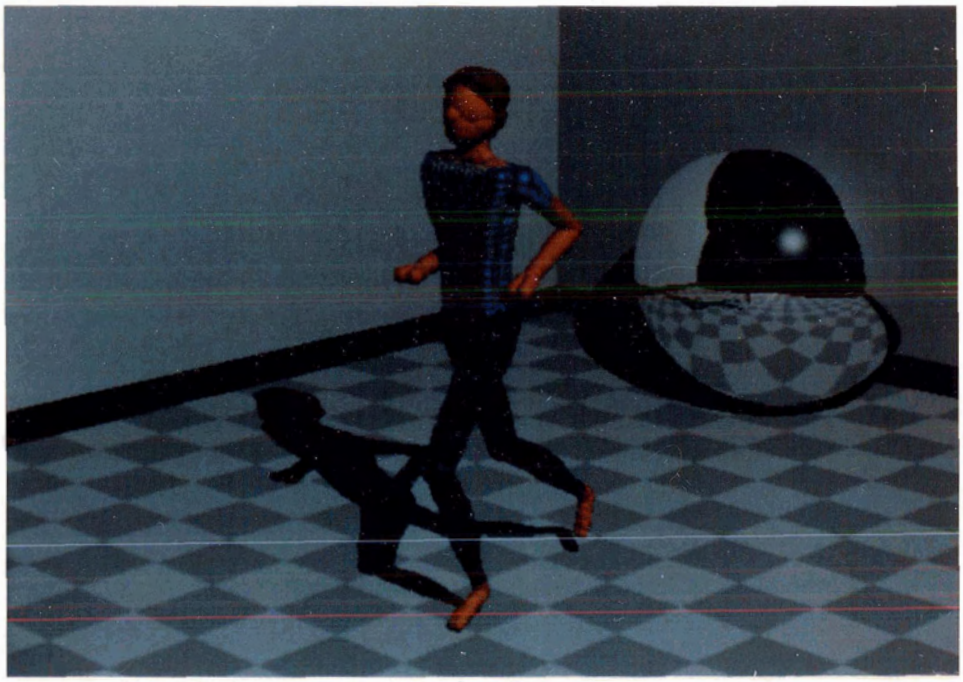

Figure 1-4: Sample scene for analysis

Varying the size of the parameters can significantly increase the number of intersection calculations that must be performed. For example:

- doubling image resolution R_0 to 1024×768 increases I_t by a factor of 4.
- adding 2 more direct light sources to the scene doubles I_t.
- doubling the number of objects in the scene also doubles I_t.

This analysis was based on the standard algorithm whereby all rays are intersected with all objects. Fortunately, many modifications have been proposed to the algorithm to increase its performance. **These improvements are discussed in the next chapter.**

Chapter 2

ALGORITHM IMPROVEMENTS

Whitted [WHIT80] has stated that intersection calculations can account for up to 95% of the rendering time Using the standard recursive algorithm, the work due to intersection calculations is expressed as *number of rays* x *number of objects* To reduce the time to accomplish a task one can either work faster or one can work more efficiently. Working faster means using faster computers special-purpose processors or specialized architectures These are issued discussed in the next chapter. Working more efficiently means reducing the number of intersection calculations by either reducing the number of rays spawned or by reducing the number of objects that must be intersected, or both. In this chapter, proposed improvements to the standard algorithm are discussed.

2.1. Reducing Rays

The number of rays spawned during the rendering of an image is dependent on many factors such as the number of pixels to be traced, the number of lights, the amount of empty space in the scene and the density of reflective and transparent surfaces. These factors are outside the control of the renderer. Where the renderer has control over the number of rays is in the process of spawning secondary rays

Adaptive tree depth proposed by Hall [HALL83] is aimed at controlling the depth of a pixel's intersection tree Before spawning a ray, the maximal contribution that the ray could potentially make to the final pixel value is calculated. If this contribution is below a

pre-determined threshold, the ray is not spawned. Hall has shown that even in highly reflective scenes such as a room of mirrors, the average tree depth was 1.71.

Assuming an average tree depth of 1.71 in the analysis discussed in the previous chapter both the number of rays traced and the intersection time would be reduced by 62%.

2.2. Reducing Objects

Reducing the number of objects with which a ray must be intersected holds the greater potential for increasing performance. Rather than doing a blind search through the entire list of objects techniques have been proposed to partition the objects or the scene to permit a more efficient search. The objective is to determine the subset of objects which are spatially close to a given ray such that the chances of the ray intersecting any of these objects is greater. In all techniques discussed below, the data organization particular to each is created as a pre-processing step. The time penalty for pre-processing is typical less than 8% of the new image generation time which is, in turn, significantly less than the standard algorithm time.

2.2.1. Bounding Volumes

Objects that require complex intersection calculations, such as is needed for fractal or spline surfaces, can be enclosed in a bounding volume, such as a sphere or a rectangular parallelepiped this results in a much simpler intersection calculation that will potentially save time If the ray does not intersect the bounding volume, then there is no need to execute the test with the complex object. Similarly, if one has built an object from a collection of objects, for example, the collection of spheres making up the forearm of the jogger in Figure 1-4 , this logical collection of spatially related objects can also be enclosed within a bounding volume to save on intersection calculations.

The concept of bounding volumes, [CLAR76], [WHIT80], involves enclosing a complex object or a collection of objects as tightly as possible within a volume which is simple to intersect. If a ray is tested for intersection against this volume and fails, the result is that the enclosed object or objects are efficiently eliminated from the intersection calculation.

Figure 2-1 shows a 2-D view of a collection of spheres bounded by a box. Ray *a* intersects the volume and so must be tested against every enclosed sphere; ray *b* fails the intersection test with the volume thus avoiding 12 intersection calculations with the enclosed objects.

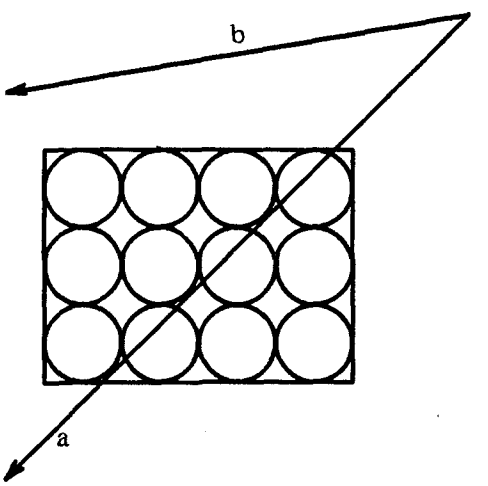

Figure 2-1: Example of a bounding volume

The decision on how to group objects and on which bounding volume to choose is largely in the hands of the user who models the scene. Weghorst et. al. [WEGH84] have done some work on the automatic selection of bounding volumes using the criteria of void area and a total cost of intersection test function. Both of the criteria are ray dependent and thus scene dependent.

At this stage we have a collection of bounding volumes. The next step would be to have a process whereby only bounding volumes lying along a ray's path are tested for intersection.

2.2.2. Hierarchical Data Description

From a collection of bounding volumes, a hierarchical data description, [CLAR76], [WEGH84], can be built using a similar approach as for the definition of bounding volumes. Collections of bounding volumes that are spatially close can be enclosed by a larger bounding volume and so on, until the whole scene is enclosed. The result is a tree where the root node is this volume, the interior nodes are bounding volumes enclosing bounding volumes and the leaves are bounding volumes enclosing objects. Again the choice of volume and the grouping of the volumes are largely defined by the user during the modeling process

The purpose of the hierarchy is to rapidly eliminate bounding volumes and objects from the intersection calculation. When a ray is spawned, it is assumed to always intersect the root volume. It is tested against the second level bounding volumes. If a volume is intersected, a recursive descent of the hierarchy begins. The saving occurs because a bounding volume is tested for intersection if and only if its parent volume has been intersected by the ray. The hierarchy is pruned down to the leaf level. Figure 2-2 shows a 2-D representation of a scene with its corresponding hierarchy.

Weghorst et. al. have shown savings of 12% to 55% over the use of bounding volumes only. Our own results have shown that the use of both bounding volumes and a hierarchical data structure decreases rendering times by up to 95% over the standard algorithm.

The efficiency of using bounding volumes with a hierarchical data structure is largely in the hands of the user. The depth of the data tree, the number of children per node, the number of objects per bounding volume are critical to the performance of the algorithm. This dependence may seem to be a liability but it may also be an advantage for the following reason. The performance of any ray tracing algorithm is dependent on the scene model. A user with a good understanding of the use of bounding volumes can thus tailor these volumes for efficiency.

2.2.3. Octree Subdivision

Glassner [GLAS84] has proposed a technique based on octrees for sub-dividing the modeling space into a hierarchical structure of subvolumes. Octrees allow dynamic recursive sub-division of the modeling space until each subvolume or *voxel* satisfies the termination condition. The condition or threshold is designed to to ensure that each voxel represents a uniform amount of work. The measure of work here is the number of objects that are wholly or partially contained in the voxel. The resulting voxel data organization allows the direct identification of the voxels lying along the ray's path.

The recursive sub-division of voxels begins by defining a cube which completely encloses the scene. This cube is the root of the hierarchical subvolume structure. The cube is divided into eight cubes or voxels each of which is tested for the termination condition. If a voxel fails the test, it is in turn subdivided and so on until all voxels have no more than the threshold number of objects. An example of the sub-division is shown in Figure 2-3

Unlike the hierarchical data description described above, the hierarchy of voxels is in itself unimportant to the rendering process. There is no need to traverse a data tree. Only the

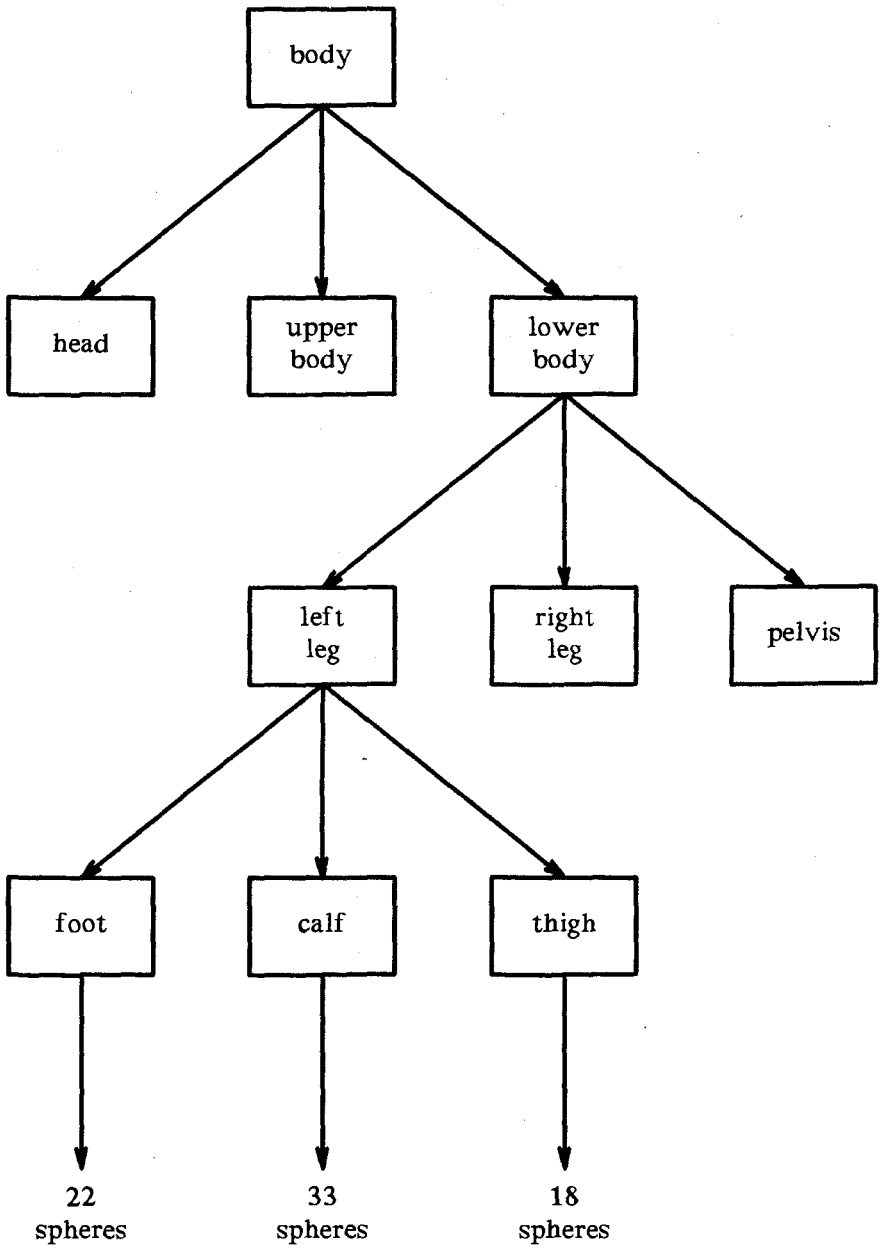

Figure 2-2: Example of a hierarchical data description

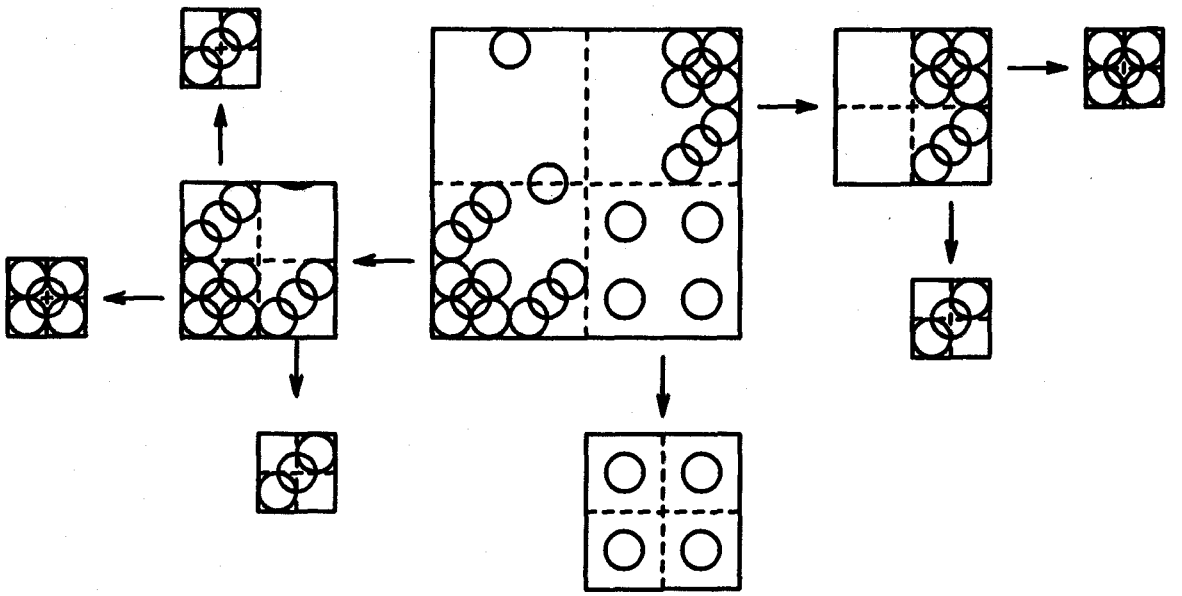

Figure 2-3: Examples of voxel sub-division

leaf voxels are kept along with their associated object lists. Using this structure, Glassner has proposed a method of quickly computing the transfer of a ray from one voxel to another. When a ray is spawned, its first voxel intersection is computed. From there, if no intersections are found within the voxel, the next voxel along the ray's path is computed and the intersection test begin with it's children. Voxels are examined in same order that the ray encounters them in the modeling space. If an intersection is found in the current voxel, the ray need not be traced any further.

Published results using this approach have shown decreases in total rendering time of 70% to 90% compared to the standard algorithm.

This approach to eliminating object intersections is straight forward and elegant. It allows one to intersect only those objects associated with the voxels lying along a ray's

path. It also gives the ray access to voxels in order of increasing distance, allowing termination of the tracing process if an intersection in found in the current voxel. However, there are potential weaknesses. The first is that the voxel threshold is based on the number of objects as opposed to the computational work required to process the voxel. A complex object could unbalance a voxel. Secondly, an object could span several voxels, necessitating several ray-object intersections for the same ray and object. Again, with complex objects, this could be a significant drawback.

2.2.4. Modeling Space Subvolumes

Another approach to reducing the number of ray-object intersections is modeling space subdivision [ULLN83] and [CLEA83]. Although developed primarily for parallel processor implementation, the technique itself is presented here within the context of a sequential algorithm The concept is similar to octree subdivision in that the modeling space is divided into subvolumes where each subvolume has a list of objects that it wholly or partially contains. The difference is that the subvolumes are geometrically uniform subdivisions in two or three dimensions and are not recursively subdivided. The process of tracing a ray is similar to the process used with the octree subdivision technique.

Unfortunately, in addition to having the same weaknesses as octree subdivision, modeling space subvolumes have an added disadvantage - there is no attempt to balance the workload associated with each subvolume. As mentioned, the algorithm's strength lies in its adaptability to parallel processing and, as such, it is discussed within that context in the next chapter.

2.2.5. Light Rays

The last technique discussed here has more to do with how a light ray is processed than with a more efficient search. The purpose of light rays is to determine if a direct light source is visible to the origin of the ray. If the light ray intersects *any* surface, the direct light source for which the ray was spawned does not contribute to the colour of the point and can be ignored. The search through the object list can then be stopped on finding the first intersection. Since light rays can account for 50% or more of the rays spawned, the potential reduction is significant.

2.3. Discussion

Improvements to the standard algorithm have been presented. Two techniques, adaptive tree depth and light rays, can be incorporated in any algorithm. On the other hand, a choice has to be made between octree subdivision or bounding volumes with hierarchical data structure. Unfortunately, published results do not use the same scene models, resolutions, shading models, performance measurements, and computers, making absolute comparisons difficult. Until someone publishes a good comparative study, the choice of algorithm must be made on different criteria, eg., which one has the least significant weaknesses.

Chapter 3

ARCHITECTURAL PERSPECTIVE

Ray tracing machines can be loosely classified into 3 classes based on the aspect of concurrency they exploit. The *intelligent pixel* machines exploit parallelism by distributing local intelligence to each pixel (or a group of pixels). This is possible since pixel computations are independent of each other. In the *intelligent object* class, processing power is allocated to each object. Thus, for a given ray, each object computes intersections in parallel. The *intelligent volume* machines subdivide 3D modeling space into subregions and allocate processing power to each region, which is now solely responsible for the objects that lie within its own volume.

In this chapter we shall the examine architectures that have been proposed or built specifically for ray tracing. We shall conclude with a discussion of the relative merits and drawbacks of the various architectures proposed.

3.1. Ullner's Machines

Ullner [ULLN83], in his doctoral thesis, proposes three different machine organizations. In the first approach, the intersection computation itself is massively pipelined to provide high throughput In the second approach, which would fall under the *intelligent object* classification suggested above, each object is processed simultaneously. Finally, in the third approach, objects are separated into disjoint regions, and these regions are processed independently, thus following the *intelligent volume* approach.

3.1.1. The Ray Tracing Peripheral

As observed by Whitted and Rubin [WHIT80, RUBI80], most of the time in a ray tracing algorithm (70-90%) is spent in finding ray surface intersections. Therefore, if these intersection computations could be cast into hardware, one could significantly reduce the running time of the ray tracing algorithm.

Ullner proposed a ray tracing processor which acts as a peripheral to a host computer. The host computer fires rays at the peripheral which in turn returns the closest polygon intersected along with the intersection information. The ray tracing peripheral has its own copy of the scene model which besides reducing the load on the host's memory, also permits the model to be organized in a way that is suitable for intersection computation.

At the topmost level the ray tracing peripheral is organized as a three stage pipeline, see figure 3-1, each of which may be internally pipelined. The first stage fetches successive polygons from a scene model memory and passes their representations to a second stage, which performs the actual intersection. The third stage examines each new intersection and discards all but the the one closest to the origin of the ray. Note that the ray must be intersected against each polygon in the scene model before the closest one can be determined. Since most of the work must be done by the intersection stage, it may internally be pipelined, as shown in figure 3-2, to increase its performance. Applying stepwise refinement we can further internally pipeline each of the stages shown in figure 3-2 until we reach the level of the actual operators implementing the arithmetic.

Two potential problems need to be addressed at this point. In order to keep the pipe full, the polygon parameters used must be accessed in parallel. This is resolved by storing each of the twenty polygon parameters in one of twenty independent memories so that all

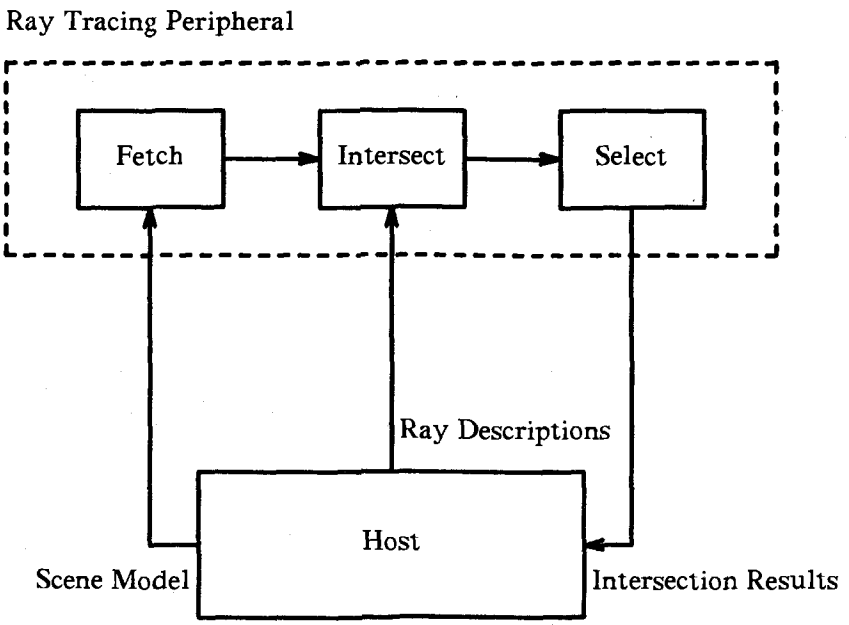

Figure 3-1: The three major pipeline stages in the ray tracing peripheral

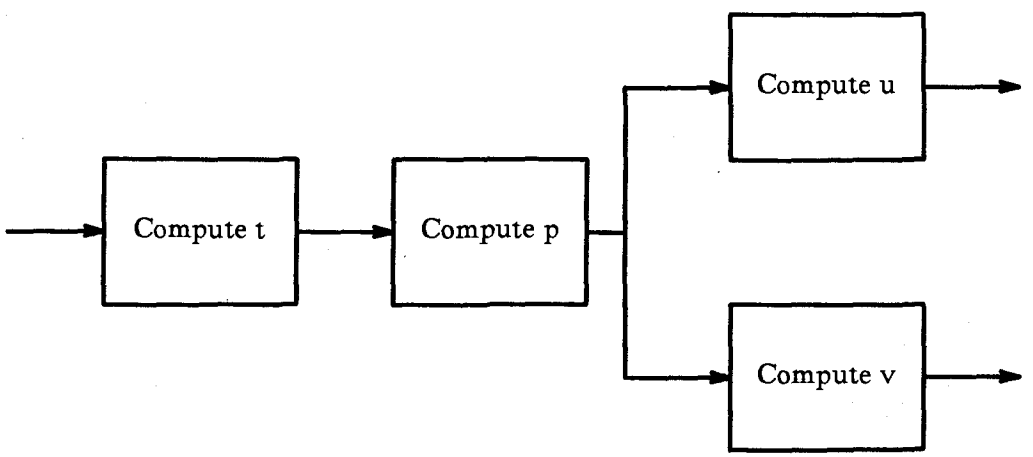

Figure 3-2: Pipeline stages within the Intersection Processor

may be accessed simultaneously. The second point is that an exception, such as in the divide operation, may be generated within the pipe, since the results may be undefined for some values of inputs. To resolve this Ullner associated a validity bit with each intermediate result flowing through the pipe. By convention, operations in the pipeline will always produce a result, but will mark that result to indicate its validity. Although later stages will accept these invalid values as if they were meaningful, the fact that their own results are invalid will be reflected in the validity bit of the output. The last stage in the pipeline takes into account the validity bit in determining the closest intersection.

All of Ullner's machines use floating point number representation which has a far greater dynamic range than fixed point numbers, freeing the user from having to pay much attention to scaling. Analysis of the ray tracing peripheral assumes that all the data operators in the pipeline are implemented using a parallel multiplier manufactured by TRW which is capable of producing a 48 bit product from two 24 bit operands in a maximum of 285 ns. Using the TRW multiplier, and a few "glue chips", a floating point multiplication takes about a third of a microsecond, but the other floating point operations cannot be completed so quickly. Each one of these operations may however be pipelined to operate at the same rate. Thus using this fully pipelined arithmetic the complete peripheral can produce three results every microsecond.

Using the above metric, we could make some estimates for the time required to generate a picture using the ray tracing peripheral. Assuming a scene model consisting of a thousand polygons, it would take a third of a millisecond to intersect a ray with each of these surfaces. In an image with 512 X 512 pixels of resolution, it would take a minute and a half to trace one ray per pixel. Of course, the number of rays increases if shadows are to be modelled and antialiasing is to be performed. Note that the time is linearly dependent on the number of polygons in the scene.

3.1.2. The Ray Tracing Pipeline

The ray tracing peripheral described earlier was not very extensible; it could not be easily enhanced to accomodate a more complex scene. The ray tracing peripheral has a single but fast intersection processor, but the intersection process has to be repeated for each polygon. Consider the other extreme now. If we had a less complex, and therefore slower, intersection processor, we could have many more of these processors working in parallel to achieve similar performance. The obvious advantage would be extensibility. The greater the number of these intersection processing units, which could be implemented as custom VLSI processors, the shorter would be the time for a more complex scene. Ideally, every object in the scene model could be attached to one of these processors typifying the *intelligent object* paradigm.

Based on the above principles, Ullner proposed the ray tracing pipeline which comprised intersection processors strung together to form the pipeline shown in figure 3-3. Each processor stores the description for a single polygon and it passes the description of rays through its input and output ports. On receiving a ray description the processors determine whether that ray intersects its stored polygon, and if so, locates the intersection point. Each ray is represented by a descriptor which has a field for the identity of the closest polygon encountered so far, and another for the *t value* of the polygon. The *t value* is initialized to infinity before entering the pipe. As it flows through the pipeline, each processor, on finding an intersection compares its *t value* with current *t value* in the descriptor field. If it is less, then that processor's polygon must be closer, and hence it swaps the identity of the polygon and the t value before passing it on through the output port to the next processor. Finally, when the ray descriptor leaves the pipeline it contains the identity of the closest polygon and corresponding t value.

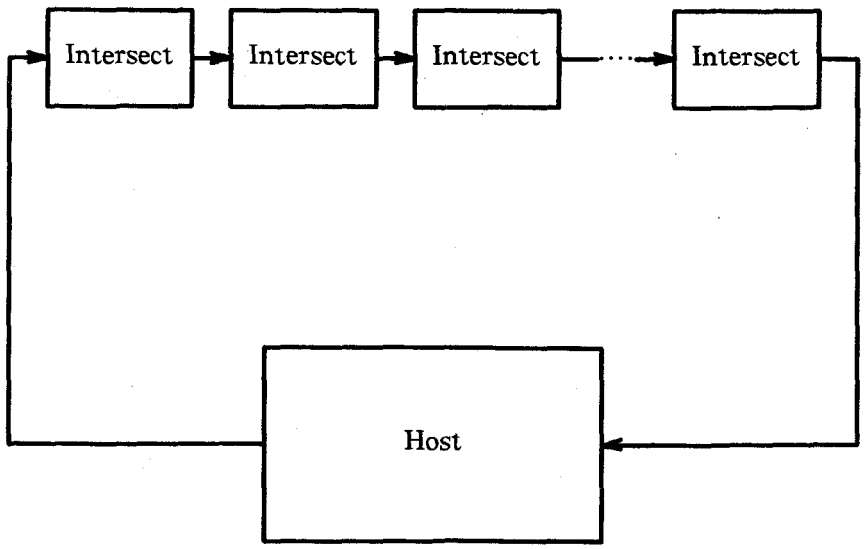

Figure 3-3: The Ray Tracing Pipeline

Since the ray tracing pipeline assumes the availability of low cost custom designed intersection processors, it would not be feasible to devote substantial chip area required to implement parallel multiplication circuitry to match the performance of the TRW multiplier used in the ray tracing peripheral. The alternative is to use a space effective, but slower, shift and add multiplier. Ullner estimates such an multiplier would perform a full 32 bit floating point multiplication in five microseconds, and also shows how other floating point operations can be implemented in the same area and speed.

Based on the above, we can conclude that the ray tracing pipeline can complete a ray tracing computation every five microseconds. Since Ullner estimates, for bit serial

communication, the transmission time to be roughly five microseconds, we are still looking at a ray being processed every five microseconds. For a machine with a thousand processors, the latency would be 5 ms., and a 512 X 512 pixel image could be generated in 1.3 seconds assuming one ray per pixel.

3.1.3. The Ray Tracing Array

In the ray tracing array, a three dimensional grid is superimposed on the modelling space to section off the volume into a collection of subvolumes, each one of which has, at least in concept, a dedicated processor typifying the *intelligent volume* approach. Each of these processors is responsible for maintaining the surface models in its own subvolume, as well as for computing intersections of these surfaces with the rays passing through the subvolume. With such an arrangement one would expect a 3 dimensional lattice of processors, each connected to its six neighbouring processors. However, the cumbersome nature of wiring entailed by such an organization, acts as a major deterrent. Ullner overcame this problem by organizing the machine as a 2 dimensional array of processors with the third dimension of the partitioning grid simulated within each processor in the array. This structure allows each processor to communicate with its four neigbouring processors, as shown in figure 3-4. Each processor is also assumed to be a general purpose computing element since each processor should now be capable of carrying out shading computations, which in previous architectures were carried out in the host. Each processor also has some special purpose intersection hardware to aid in intersection computation.

The processors communicate with each other through messages. Each processor is responsible for a block of pixels corresponding to its position in the array and has an independent frame buffer used to store the pixel intensities. The different fields of the ray

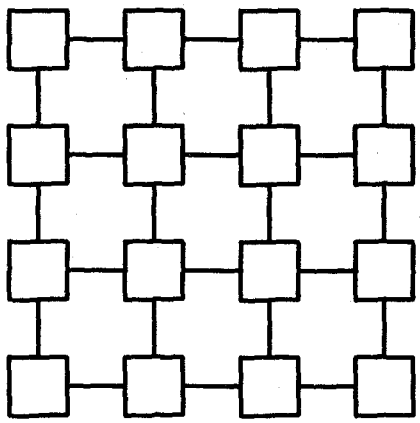

Figure 3-4: Organization of processors in a 16 processor ray tracing array

k	Message type (e.g. vision, shadow, etc.).
(r,c)	Row and column of pixel for this ray.
ro	Origin of this ray.
rd	Direction of this ray.
c	Color contribution of this ray.

Figure 3-5: Fields of a ray message

message are show in figure 3-5 Processors create initial ray messages for pixels that lie within their portion of the frame buffer. The processor then computes the closest subvolume which the ray enters, and then passes the ray message in the direction of the processor responsible for that subvolume. On reaching the destination processor, the ray is tested for intersection against all the objects within the subvolume. If no intersection is found then the processor incrementally computes the next closest subvolume which is handled by one of the four adjacent processors, and sends the ray message in that direction. If an intersection is found, a result message, which contributes to the intensity of its originating pixel, is passed off to the processor responsible for that pixel. Any secondary rays such as reflected, refracted or light rays are passed off to appropriate subvolumes for further intersection tests.

Cleary, et. al. [CLEA83] also proposed a similar processor array for ray tracing. They considered both square arrays and cubic arrays, and found that, in general, square arrays perform better than cubic arrays. A machine based on a 10 x 10 square array is currently under construction at the University of Calgary.

3.2. Dippe's Parallel Architecture

Mark Dippe & John Swensen [DIPP84], proposed an architecture for ray tracing which is quite similar to the ray tracing array proposed by Ullner, thus belonging to the *intelligent volume* family. The major difference between the two is that Dippe's parallel architecture allows for the subdivision of object space to be adaptively controlled, in order to maintain a roughly uniform load amongst the different processors. This turns out to be a serious drawback in Ullner's ray tracing array where no attempt was made to address the issues of uniform load distribution over the subregions. Uneven object distribution amongst different subregions can lead to load disparities between processors, causing computing

power to be wasted. Therefore the ability to adaptively redistribute over time is crucial because load distributions are extremely difficult to calculate *a priori*, and hence must be done dynamically during the actual execution of the ray tracing process.

Since the operation of this parallel architecture is very similar to the ray tracing array, we shall concentrate on the dynamic load distribution aspect of this organization. The three dimensional space of the scene to be rendered is divided into several subregions which are initially assigned volumes more or less uniformly, and object descriptions are loaded into the appropriate subregions. As computational loads are determined, the space is redistributed among the subregions to maintain uniformity of load. Unlike the straightforward orthogonal subvolumes in Ullner's architecture, Dippe considered several different shapes for subregions. The choice of a subregion shape is influenced by the following criteria:

1. the complexity of subdividing the problem e.g. intersecting objects or rays with the boundaries.
2. the ability to subdivide space without splitting objects, and
3. the uniformity of the distributed loads attainable with the shape.

A strong candidate based on the abovementioned criteria would be "general cubes", which resemble the familiar cube, except they have relaxed constraints on the planarity of faces and on convexity. General cubes allow the most local control of subregion shape at the cost of slightly higher complexity of boundary testing.

The load information is shared among the neigbouring subregions, and this allows relatively more loaded subregions to reduce load by adjusting their boundaries. The load metric is primarily determined by the product of

1. number of objects and their complexity, and

2. number of rays

Load is transferred by moving corners of a subregion. Once the new position for a corner of a subregion has been determined, object descriptions and other information are redistributed to reflect the new subdivision.

Due to the subdivision, a speedup of the order of $O(S^{2/3})$ is expected by the authors, where S is the number of subdivisions of the object space. The parallel architecture is estimated to be three orders of magnitude faster than the standard algorithm with 125 computers working in parallel.

3.3. The LINKS-1 Multimicrocomputer System

LINKS-1 [NISH81] was an experimental machine which was built and tested at Osaka University in Japan. The system consists of 64 unit computers which are interconnected with a root computer such that a number of unit computers constitute a pipelined computer and such pipelined computers work in parallel, all controlled by the root computer. The number and length of each pipeline can be controlled dynamically, although it is not readily apparent how this dynamic reconfiguration would be useful. On the other hand the organization is general enough to be used for other image creation applications by means of more sophisticated parallel processing schemes which utilize different numbers of pipelines, perhaps with different lengths. Intercomputer program/data transfer is greatly facilitated by the use of a device called the intercomputer memory swapping unit (IMSU). LINKS-1 permits neighbouring unit computers to exchange data/programs using IMSU, and also between each unit computer and the root computer. There also exists a slow serial link between each unit computer and the root computer.

The root computer distributes the programs and data to be executed to the unit computers and the results are collected by the data collector. Each unit computer comprises five units:

1. the Control Unit for data transfer and communication control,
2. the Arithmetic Processing Unit for floating point calculations,
3. the 1Mb Memory Unit,
4. the I/O unit to be used as an outlet for debugging and monitoring,
5. the Intercomputer Memory Swapping Unit (IMSU).

The IMSU has two memory areas which are connected to a pair of control units through a bus exchange switch. Each of the control unit works independently on a memory area, and upon finishing they send a bus exchange signal which connects them to the other memory area.The IMSU is used to exchange program/data both between the root computer and the unit computers and also between two adjacent computers.

3.4. Discussion

Both the ray tracing peripheral and the ray tracing pipeline are, in a way, brute force approaches to the ray tracing problem, since they attempt to intersect every ray with every polygon. As noted in earlier chapters, techniques such as object space subdivision and bounding volumes can be used to significantly minimize the most computationally expensive operation — the ray surface intersections. The ray tracing peripheral, however, can be modified to use object space subdivision. The basic idea here is to superimpose a three-dimensional grid on the object space. The objects are then partitioned into these subvolumes. An extra stage is added to the pipeline which computes the subvolume which the ray intersects and passes the descriptor addresses of the polygons residing in the

subvolume onto the next stage. Thus, the subsequent stages only have to compute intersections with a small number of polygons. No such arrangement is possible with the ray tracing pipeline since a separate pipe would be required with each subvolume.

The ray tracing pipeline is ostensibly fast, but on careful observation one quickly realizes that no general purpose host could keep up with it since it is unreasonable to expect a host to generate ray descriptions at this rate and deal with responses in the same time. Of course, one can design a special purpose host, sacrificing the flexibility offered by a general purpose host. It is also impossible for the ray tracing pipeline to process a scene with more objects than the number of processors in the pipeline. Note that this does not pose a problem for the peripheral since in the worst case all that needs to be done is to increase memory size. In case of the ray tracing pipeline, however, it becomes infeasible to increase the number of processors after a certain point.

Ullner's machines assume convex quadrilaterals as the basic modelling primitive. To achieve maximum performance, all intersection processors are dedicated to ray intersections with polygons. In computer graphics, however, it is often advantageous to model with alternative surface representations, such as bicubic patches, splines, quadric surfaces etc. The dedicated intersection processors are incapable of performing these intersections. On one hand, it appears in order to accommodate a variety of modelling surfaces, the intersection processors should be general purpose with fast floating point hadware to boost performance. On the other hand, we could tesselate most modeling surfaces into polygons and continue using dedicated intersection processors. Interestingly enough, there are devices available, such as the Weitek Transformation Engine [WEIT85a], which perform the tesselation functions with great speed.

The ray tracing array is probably the most promising approach of the three machines proposed by Ullner. Its chief drawbacks stem from the straightforward orthogonal subdivision of object space, which can cause immense disparity in object distribution among the subvolumes. Dippe's architecture takes care of this problem by using an adaptive subdivision approach. Also, for some choices of viewing position, not all processors are equally busy.

The Links-1 has a topology that allows work to be distributed by the root computer so that it can be performed independently in parallel, or pipelined from neighbour to neighbour, or some combination of both. This allows a variety of image creation algorithms to be used. But, the connection topology is restricted enough that any situation which demands substantial communication amongst the various unit computers would be almost impractical.

Chapter 4

A 3-TASK RAY TRACING ALGORITHM

In the previous chapters we discussed approaches for improving ray tracing performance by reducing the amount of computation and by increasing the speed of computation. As demonstrated in the modeling space subvolume approach, algorithms can be designed that directly map onto system architectures.

In this chapter we describe our modified ray tracing algorithm which maps directly onto a pipelined parallel processor architecture. To reduce the number of intersection calculations, our algorithm is based on bounding volumes and the hierarchical description of data. This approach also allows the tracing of a ray to be divided into three balanced tasks that map onto the pipeline architecture. In addition, the potential for parallelism lies in image space subdivision where a pipeline can independently compute the value of a given set of pixels.

4.1. Definition of Terms

The following definitions are for terms used in this and following chapters. Some of the terms are similar to those used in [WEGH84].

contribution factor
: factor which determines the contribution made to the pixel by the intensity found at the end of the ray.

data tree
: the hierarchical description of the scene; its non-terminal nodes are parent shells and its terminal nodes, leaf shells.

initial ray
: a ray originating at the eye and passing through a pixel on the image plane.

leaf shell	a shell which encloses primitives; whose children are primitives.
light	a geometric entity with an associated set of emittance characteristics.
light ray	a ray spawned on intersecting a reflecting surface in the scene; its origin is the intersection point and its direction is toward a specific light.
object	a geometric or procedural entity with an associated set of surface characteristics reflecting and possibly transmitting light.
parent shell	a shell which encloses shells; whose children are shells.
prim processor	performs the ray-primitive intersections.
primitive	an object or a light.
ray	a vector with a specific origin and direction.
reflected ray	a ray spawned on intersecting a reflecting surface in the scene; its origin is the intersection point.
refracted ray	a ray spawned on intersecting a transmitting surface in the scene; its origin is the intersection point.
scene	the uppermost parent shell in the hierarchical description; it has no parent shell.
shade processor	spawns initial and secondary rays; also computes the contribution a ray makes toward the final pixel value.
shell	a bounding volume.
shell processor	performs the ray-shell intersections.
t-value	a parametric value that defines a point on a ray where the ray intersects a surface.

4.2. Overview

Before delving into the details, we present a brief overview of the algorithm. An initial ray is spawned. This ray is tested for intersection against the nodes of the data tree in a recursive depth-first descent. If a parent node is intersected by the ray, all its children are in turn tested; if not, that branch of the tree is ignored. A list of all leaf shells intersected is generated and sorted in order of increasing t-value. The next step is to determine the closest primitive intersected. Beginning with the leaf shell closest to the origin of the ray, its child primitives are tested for intersection. If no intersection is found, the child primitives of the next closest leaf shell is tested and so on.

When an intersection is found, secondary rays are spawned. Using the surface characteristics associated with the intersected surface, the contribution each secondary ray makes to the final pixel value is computed and tagged onto the ray. Secondary rays are then processed in the same fashion as the initial ray. When all rays spawned for a pixel have been traced, the pixel value calculation is complete.

4.3. Features

Several features of our algorithm are important to its eventual mapping onto an architecture.

4.3.1. Data Tree

The data tree has two restrictions. The first of these is that all primitives must be enclosed within a leaf shell, either individually or within a collection of other primitives. Secondly, a parent shell can only have shells as children; a leaf shell can only have primitives as children.

4.3.2. Shell Shape

So far we have talked about shells without making any specific reference to the shape of the shells. The shape of the shell is an important issue, as discussed in [WEGH84]. We explored two of the possible alternatives for shells - spheres and orthogonal boxes. Orthogonal boxes have sides parallel to the axes of the modeling space coordinate system. In general, orthogonal boxes serve as better shells than spheres for the following reasons:

- In general, orthogonal boxes have less void area than spheres; they enclose their primitives more tightly. This increases the probability that a ray will intersect an enclosed primitive if it intersects the shell.
- The ray-shell intersection test is faster to compute. Note that if we only needed to know whether a ray hits or misses a shell, then spheres would be better since they require fewer floating point operations. If the exact point of intersection is also desired, then the intersection of a sphere, which requires computation of a square root, is slower.

Table 4-1 shows results that support the argument regarding shell shapes. The total rendering time is tabulated for a sample scene using the two shapes.

SPHERES	4162.01 secs.
ORTHOGONAL BOXES	2727.29 secs.

Figure 4-1: Total time taken for rendering a sample scene using spherical shells and orthogonal box shells

Another possibility is to use randomly oriented boxes, which potentially have less void area than orthogonal boxes. However, more overhead is associated with these boxes. The ray has to be transformed into the coordinate system of the random box and more data

(the transformation matrix) must be stored. As we shall see later, in the context of our proposed architecture, the extra computations and the larger size of the shell data set could prove to be costly. Hence, orthogonal boxes represent a compromise between architectural demands and intersection efficiency.

4.3.3. Simplified Shader

The algorithm used a simplified version of the Hall shading model described in Chapter 1. The current algorithm does not trace rays through transparent surfaces. Fresnel reflectance and transmission curves and distance factors are also not implemented. Intensities and reflectance characteristics are represented using RGB triplets (a value for each of the primary colours - red, green and blue). The same RGB triplet is used for both specular and diffuse reflections. Using terms defined in Figure 1-2, our model is as follows:

$$I = \sum_{j=1}^{l} [k_d(\overline{N}\cdot\overline{L}) + k_s(\overline{N}\cdot\overline{H})^n] R_d I_j + k_s R I_r + k_d R I_a$$

Our algorithm and proposed architecture do not limit the complexity of the shading model. The reason for its simplicity has more to do with our emphasis on architecture.

4.3.4. No Intersection Tree

Although useful for describing the concept of ray tracing, intersection trees are not necessary in practice. Secondary rays are spawned to determine the intensities of various sources of illumination. The maximum contribution to the final pixel value that can be made by the intensity of a source of illumination can be computed. This contribution

factor is calculated from the intersected surface characteristics and the intersecting ray factor. If a source of illumination does contribute, its intensity is multiplied by the contribution factor and the result added to the pixel value. To keep track of which ray belongs to which pixel, each ray is tagged with the pixel coordinates.

The advantage of this approach [ULLN83] is in removing the memory requirements and computation overhead associated with building and traversing intersection trees. This is especially important in the context of a VLSI processor pipeline.

4.3.5. Adaptive Tree Depth

Computing the contribution factor of a ray before it is traced enables us to use adaptive tree depth. If the factor is below a significant threshold, its contribution can be ignored and thus the ray need not be traced.

4.3.6. Primitives Types

Currently, the types of objects that our algorithm can render is limited to spheres and polygons. Work is currently underway to add fractals to the system. The algorithm is not really limited to those primitives and could easily be expanded to include other geometric or procedural primitives such as cylinders, cones, surfaces of revolution, prisms, and 3-dimensional curved surfaces.

4.3.7. Sorting Leaf Shells

Instead of performing a depth-first descent down to and including enclosed primitives, the algorithm initially tests only as far as the leaf shells. The intersected leaf shells are then sorted in order of increasing t-value (distance from the origin of the ray). In a strategy similar to that described for octree subdivision in chapter 2, the primitives enclosed by the

nearest shell are tested for intersection. The closest surface intersected is identified. If such a surface is found, then the search is stopped; otherwise the primitives enclosed in the next closest shell are tested. This process is repeated until either a surface is intersected or no more leaf shells are left, implying that the ray does not intersect any primitive.

Unlike octree subdivision, hierarchical data organization may not produce disjoint leaf shells, i.e., shells whose volumes do not overlap. Fortunately, the above technique can be modified for use with overlapping shells. The t-value of an intersected primitive t_p is checked against the t-value of the next closest leaf shell t_s. If $t_p < t_s$, then the primitive is the closest. Otherwise the primitives in the next leaf shell must be checked.

Figure 4-2 illustrates this point. The two shells enclose exactly one primitive each.

Primitive A belongs to shell A and primitive B to shell B. Shell A is closer than shell B to the origin of the ray, i.e., $t_{shell\_A} < t_{shell\_B}$. Hence, primitive A would be tested for intersection first. Let us assume that the ray does intersect primitive A at t_A. However, as can be readily observed, primitive A is <u>not</u> the closest primitive (t_A is not less than t_{shell_B}). The primitives of shell B have to be tested before the closest surface can be identified. Here, primitive B is the closest primitive, although shell B is farther from the ray's origin than shell A.

This technique permits the identification of the closest primitive intersected without necessarily testing all the primitives in all the intersected leaf shells. Test results from rendering the scene in Figure 1-4 show that, on average, a ray tests the contents of only 80% of the sorted leaf shells.

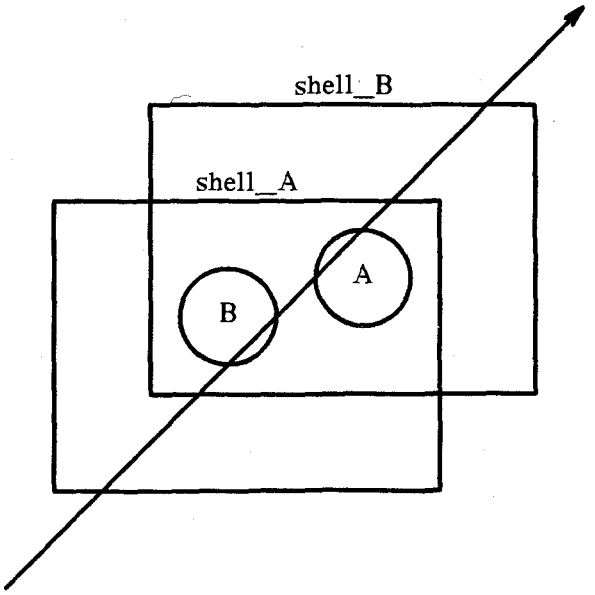

Figure 4-2: A 2-dimensional view of overlapping shells

4.4. The 3 Data Sets

Examining the data required by our algorithm, we can identify three disjoint data sets. This partitioning of the data also corresponds to the partitioning of the tasks described in the next section. The data sets are the shells of the hierarchical data description, the collections of primitives enclosed by the leaf shells and the different surface characteristics found in the scene model.

4.4.1. Shell Data

The basic element of the shell data set is the structure SHELL illustrated in Figure 4-3. The collection of shells making up the hierarchical data description is stored in an array called SHELL_ARRAY illustrated in Figure 4-4. The organization of data in this array retains the tree structure of the data tree. An entry in this array is a linked list of sibling shells, i.e., children of the same parent. The variable *leaf* indicates whether the shell is a leaf or parent shell. For a parent shell, the variable *child_index* is the index to its list of children. For a leaf shell, the variable is an index into the PRIM_ARRAY where the child primitives are stored. By convention, the index to the children of the scene or root shell is 0.

4.4.2. Prim Data

The basic element of the primitive dataset is the structure PRIM illustrated in Figure 4-5. The variable *type* indicates what type of primitive be it a sphere, polygon or whatever The variable *p* is the union structure through which the geometric description can be accessed. The variable *surface_index* is an index into the SHADE_ARRAY where the surface characteristics associated with the particular primitive are stored. The collection of primitives making up the model description is stored in an array called PRIM_ARRAY illustrated in Figure 4-6. An entry in this array is a linked list of sibling primitives, i.e., children of the same parent.

```
typedef struct  shell   {
        int             leaf;
        int             child_index;
        COORD           max;
        COORD           min;
        struct  shell   *next;
        } SHELL;
```

Figure 4-3: SHELL data structure

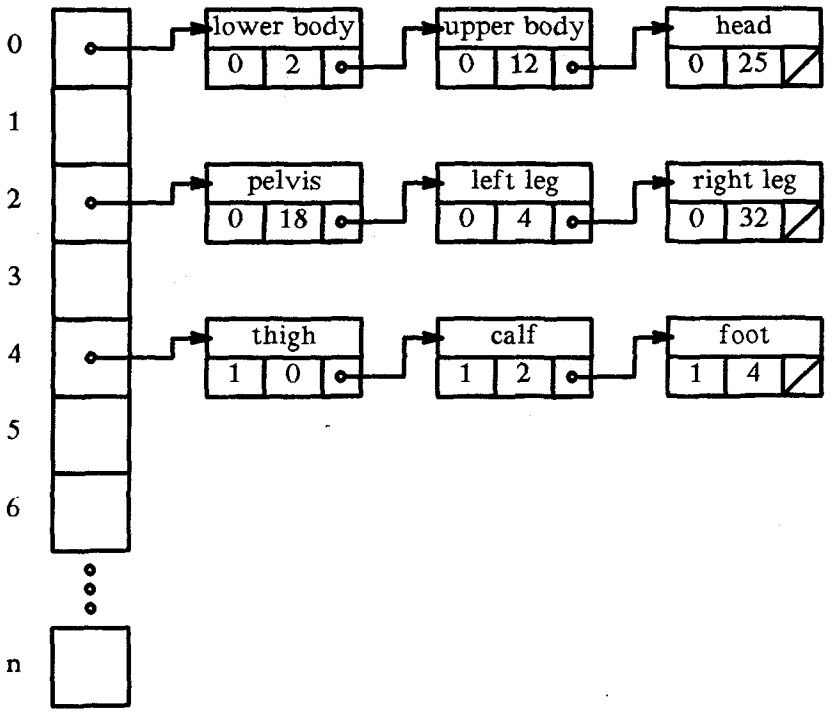

Figure 4-4: Illustration of SHELL_ARRAY

```
typedef struct  prim    {
        int             prim_id;
        int             surface_index;
        int             type;
        PTYPE           p;
        struct  prim    *next;
        } PRIM;
```

Figure 4-5: PRIM data structure

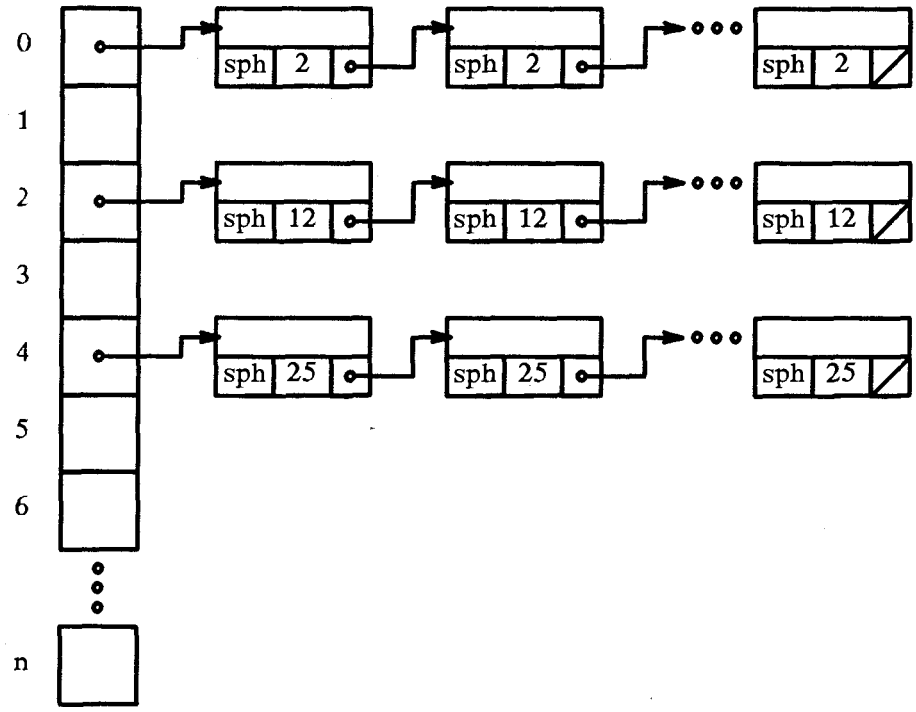

Figure 4-6: Illustration of PRIM_ARRAY

4.4.3. Shade Data

The basic element of the shade data set is the structure SHADE illustrated in Figure 4-7. Unlike the previously described arrays, the array for the shade data set is a simple array of SHADE structures. The variables *reflectance* and *transmittance* are triplets for red, green and blue values. Although the structure is designed for reflectance characteristics, emittance data can also be stored in the same structure by interpreting the *reflectance* variable as an emittance triplet and setting all other variables to 0.

```
typedef struct  shade   {
        float           k_s;
        float           k_d;
        int             n;
        RGB             reflectance;
        RGB             transmittance;
        } SHADE[];
```

Figure 4-7: SHADE data structure

4.5. The 3 Tasks

Our sequential ray tracing algorithm described above can be cleanly divided into the following tasks.

1. The first task spawns all the initial and secondary rays. It also computes the contribution factors that these rays make to the final pixel values.

2. The next task traverses the hierarchical tree with a given ray and makes up a sorted list of all the leaf shells intersected by the ray.

3. The third task intersects primitives contained in the leaf shells to compute the closest intersecting primitive.

In this section we shall outline each task's basic algorithm and the input and output data structures used by each.

4.5.1. The Shade Task

The first task, called ShadeTask, spawns rays for a given set of pixels. For each ray, an output data structure (illustrated in Figure 4-9) is filled and sent to the ShellTask described below. The variable *ray_type* indicates whether the ray is an initial, reflected or light ray. The coordinates of the pixel to which the ray belongs are found in *pixel_index* and the ray's contribution in *factor*.

When a ray returns to the ShadeTask after being traced, the combination of *ray_type* and what it hit, *hit_type*, determines the action to be taken. When a ray leaves the scene or when a light ray is blocked, the ray is ignored. Otherwise, if the ray is a light ray, the product of the intensity and *factor* is added to the pixel; if it is another type of ray, the product of the ambient intensity and *factor* is added to the pixel and new secondary rays are spawned. The algorithm is illustrated in Figure 4-8.

4.5.2. The Shell Task

The second task, called ShellTask, is outlined below in Figure 4-10. Receiving the structure SHADE_TO_SHELL as its input, the this task traverses the SHELL_ARRAY tree with the given ray. When a leaf shell is intersected by the ray, the child index and the t-value which defines the point of intersection are stored in the LeafShellList of the output data structure. When the traversal has been completed, the list is sorted on ascending t-values.

The output of the ShellTask is a structure similar to the one shown in Figure 4-11.

```
/*****************************************************************************
Function: ShadeTask
Purpose : Spawn rays and compute contribution factors according to the
          shading model.
*****************************************************************************/

ShadeTask ()

begin
if (light ray)
    begin
    if (self hit)  pixel += light intensity * factor;
    else           ignore ray;
    end
else
    begin
    if (no hit)  ignore ray;
    else
        begin
        pixel += ambient intensity * factor;
        spawn secondary rays and compute contribution;
        end
    end

if (pixel is finished)  spawn initial ray for next pixel;
end
```

Figure 4-8: The ShadeTask algorithm

```
typedef struct  {
        int             ray_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        } SHADE_TO_SHELL;
```

Figure 4-9: Output structure from ShadeTask

```
/*****************************************************************************
Function: ShellTask
Purpose : Produce a list of child indices and t-values (LeafShellList)
          of leaf shells intersected by the ray.
*****************************************************************************/

ShellTask (ix)

begin

/* Let S be the set of all shells pointed to by SHELL_ARRAY[ix] */

for each shell ∈ S
    begin
    if (the ray intersects the shell)
        begin
        if (leaf shell)  LeafShellList ← LeafShellList ∪ {child_index,tvalue};
        else             ShellTask(child_index of shell);
        end
    end

Sort LeafShellList on increasing t-value;
end
```

Figure 4-10: The ShellTask algorithm

```
typedef struct  {
        int             ray_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        LSS             LeafShellList [50];
        int             LeafShellCount;
        } SHELL_TO_PRIM;
```

Figure 4-11: Output structure from ShellTask

4.5.3. The Primitive Task

The third task, which we shall call PrimTask, receives the shell to prim data structure as input. This task executes exactly what has been described in the overlapping shell discussion above. The task proceeds to intersect primitives starting with the primitives enclosed in the closest leaf shell and stops on finding the closest primitive. It then also computes the information needed by the first task, the Shader Task, such as the surface normal at the point of intersection.

The detailed algorithm is show in figure 4-12. Note that in the actual implementation the algorithm treats different types of rays differently. For example, light rays need not find the closest intersection but any intersection will do. On the other hand, for initial and reflected rays, the algorithm goes through all the primitives in the given primitive list.

The output of the PrimTask is a structure similar to the one shown in Figure 4-13. The variables filled by the task when an intersection is found are *surface_index*, *point* that contains the coordinates of the intersection point and the surface normal at that point, and *hit_type* which describes what the ray hit.

```
/*************************************************************************
Function : PrimTask
Purpose  : To compute the nearest primitive.
Note     : 1. LeafShellList comes from the ShellTask.
           2. Indices in the set LeafShellList are accessed in order i.e.
              we get the element with the least t-value first.
*************************************************************************/

PrimTask()

begin

for each index ∈ LeafShellList
    begin

    /* Let P be all Primitives pointed to by the current index. *.

    find the nearest_primitive ∈ P;
    if (t-value of nearest_primitive
                  < t-value of next index in LeafShellList)
        begin

        /* we have found the nearest primitive */

        found = TRUE;
        break;
        end
    end

if (found) compute info (intersection point, normal, surface_index);
else       report no hit;
end
```

Figure 4-12: The PrimTask Algorithm

```
typedef struct  {
        int             ray_type;
        int             hit_type;
        PIXEL           pixel_index;
        RGB             factor;
        RAYEQN          ray;
        INTER           point;
        int             surface_index;
        } PRIM_TO_SHADE;
```

Figure 4-13: Output structure from PrimTask

Chapter 5

A PIPELINED ENGINE FOR RAY TRACING

In this Chapter, we propose a pipelined architecture, *PERT*, which executes the 3-task ray tracing algorithm discussed in the previous chapter. PERT consists of a 3-stage pipeline of processors. Each stage in the pipeline is a microcoded, custom designed, VLSI processor that greatly enhances performance. PERT forms the basic computing element of a parallel architecture for ray tracing [GAUD85], which is a multi-PERT architecture with an innovative interconnection scheme.

5.1. The Single-PERT Configuration

PERT is a pipeline of three processors connected cyclically as shown in figure 5-1. This architecture is a direct map of the ray tracing algorithm described earlier, with the three processors performing the three tasks — the ShellProcessor performing the ShellTask, the PrimProcessor performing the PrimTask and the ShadeProcessor performing the ShadeTask. The organization deviates from the classical Von-Neumann architecture, since three instruction streams are concurrently active on three independent data sets, and hence would be classified as a MIMD organization under Flynn's [FLYN66] taxonomy.

PERT can be used in 2 different configurations: a) in a *single-PERT* configuration, where each of the 3 processors has access to an independent memory module that stores the appropriate data set, and b) in a *multi-PERT* configuration that consists of an interconnection of N PERTs working in parallel. PERTs in this configuration do not have

scene data available to them in local memory, but access it from three *broadcast* buses; one for each processor within PERT. Since this thesis is primarily concerned with the design and performance of a single-PERT configuration, for the remainder of this thesis, the term PERT, should be taken to mean single-PERT configuration, unless explicitly specified. We shall briefly discuss multi-PERT configuration in section 5.2, but for a complete analysis the reader is referred to [GAUD85].

The 3 processors comprising PERT are identical internally, except for their microcode. Figure 5-2 shows the internal organization of the processors. We shall now briefly discuss the various modules comprising each processor.

5.1.1. The SJ16 Processor

SJ16 is a 16 bit microprocessor that was intended to be used as a hardware building block for multiprocessor systems [HOBS81a]. SJ16 — fabricated as a single chip VLSI processor using a 5 micron GTE ISO-CMOS process, and currently being tested at Simon Fraser University — was a natural processor choice: it was microprogrammable, it had excellent hardware features such as an ALU with a barrel shifter, and on-chip hardware stack, a register file with 32 general purpose registers, and an independent up/down counter to simplify loop handling. Since microcode development for the various task algorithms was a key issue in the PERT design, the most attractive feature of SJ16 was the microprogramming environment — the Architecture Support Package (ASP). The ASP allows higher-level microprograms for SJ16 to be written in an APL like notation called microAPL. MicroAPL code can then be translated into real SJ16 microcode by a microAPL compiler and linker. Besides microcode development, the ASP also permits emulation of hardware modules by APL functions, allowing investigation of new hardware constructions. Details of microcode development for SJ16 can be found in [HOBS82].

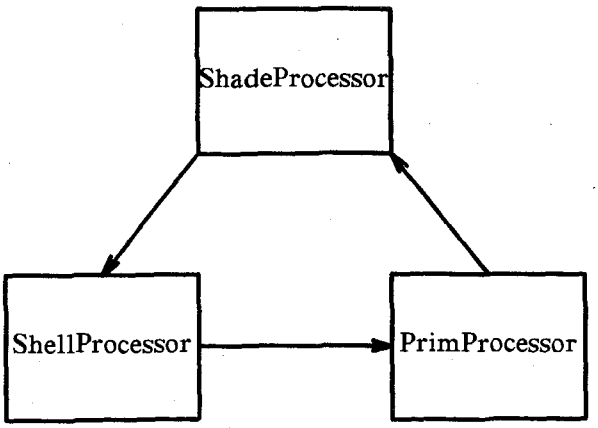

Figure 5-1: Block diagram of PERT

5.1.2. The Floating Point Unit

The floating point unit (FPU) is capable of fast execution of floating point operations. For simulation purposes, this special function unit was modeled around the Weitek WTL1164/1165 low-latency floating point chip set [WEIT85b] capable of executing floating point operations with speeds above 2.78 Mflops. Recalling the voracious appetite of the ray tracing algorithm for floating point computation, one can see that the high throughput of the Weitek chip set makes it a prudent choice.

All floating point operations on PERT are performed in single precision. Details of the internal design and simulation of the FPU are covered in section 6.2.2.

5.1.3. The Memory Module

The memory module provides independent storage for each of the three processors. The memory module is primarily used to store the data set associated with each processor. Both the ShellProcessor and the PrimProcessor also need some extra storage for global variables, stack space, etc. This extra storage required is minimal. The ShadeProcessor however, requires extra memory to be used as the frame buffer.

Reads and writes to the memory can be *streamed* — the memory controller buffers data words and hence after the first access, memory can be accessed sequentially in a single cycle.

5.1.4. Communication

The ShellProcessor and the PrimProcessor communicate with the ShadeProcessor using FIFOs. However, communication between the ShellProcessor and the PrimProcessor must be done with a dual buffer since the ShellProcessor uses one of the two buffers to fill in leaf shell ids and then performs a sort on them, which means that the PrimProcessor cannot read the shell ids on a FIFO basis but must wait until the ShellProcessor has completed its sort. With the dual buffer the PrimProcessor reads from one buffer while the ShellProcessor is busy filling the other with shell ids.

The three processors of PERT are hardware embodiments of the three tasks of the ray tracing algorithm. Since the operation of the ray tracing algorithm has been covered in great detail in chapter 4, and the operation of PERT is identical, it will not be discussed here.

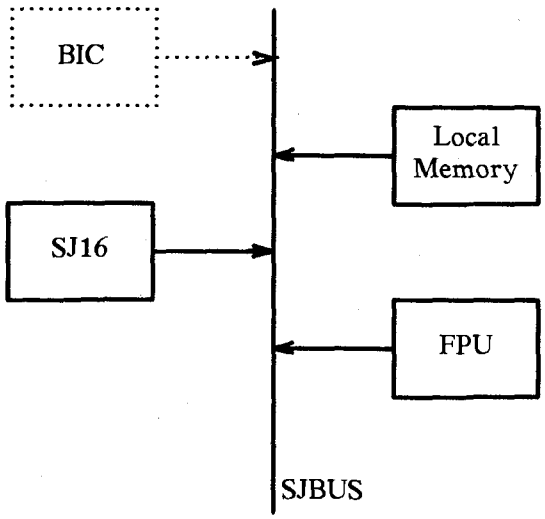

Figure 5-2: Detailed block diagram of each processor

5.2. The Multi-PERT Configuration

The basic difference between the two configurations is the way the data sets are accessed. In a single-PERT configuration this turns out to be easy since the data sets are stored in local memory and hence can be directly accessed. In a multi-PERT configuration however, we cannot afford the luxury of replicating the entire scene in each PERT, since that would be a brute force approach. What is needed is a way of allowing concurrent access, by PERTs, to a global shared memory.

5.2.1. Broadcasting

Our solution to the problem consists of having three external buses connected to each of the three processors, on which data is *broadcasted*. We draw on an analogy here to illustrate the concept of *broadcasting*. Assume we have a disk subsystem and think of the output of the read/write head as a (single line) bus to which several processors are attached as shown in figure 5-3. Let us further assume that our hypothetical disk has only one track and the read/write head, set to read mode, is permanently positioned over it. Now, what appears on the bus is a bit-stream that is repeated periodically owing to the circular nature of the track containing the bits of information. Each processor has access to any bit in the stream, but the access is sequential as opposed to being random. Thus, associated with each bit access, is a potential latency delay. We shall herewith refer to such a periodic transmission of data over a bus as *broadcasting*, the bus, which is the broadcast medium as the *broadcast bus*, and the time taken to cycle through the entire set of data as the *broadcast cycle time*.

5.2.2. System Organization

In reality, the function of the hypothetical disk is taken over by fast broadcast processors that have access to the global memory, and the processors in our analogy are really PERTs. The broadcast processors transmit data at high speeds over their broadcast buses. Speed is a critical issue here, since the slower the broadcaster, the greater would be the access latency. Each of the PERTs can now, irrespective of the others, access data off the bus as needed, without any contention for memory. Of course, for this, one has to pay a price, access delays because of latency. This could however, be minimized by the techniques discussed in the next section.

Figure 5-4 shows the overall system organization of a multi-PERT configuration. The

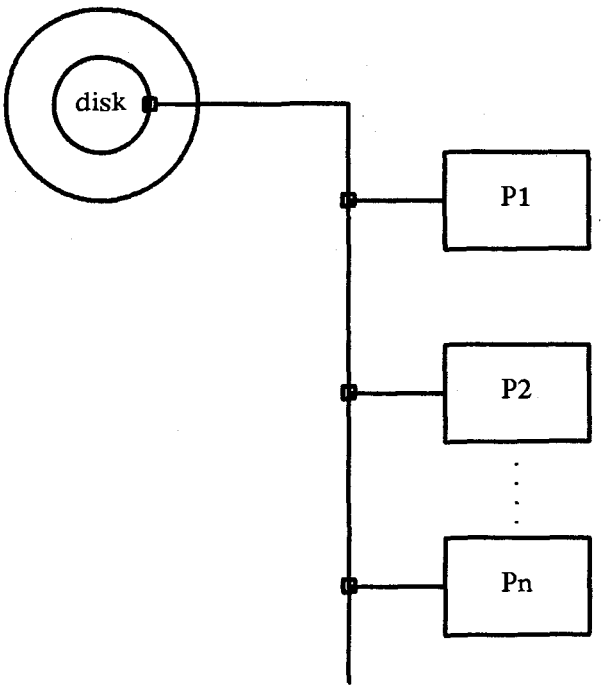

Figure 5-3: Example to illustrate broadcasting

engines. PERT1 thru PERTn, are connected to each of the three buses as illustrated. There are many ways in which distribution of work can be accomplished in a multi-PERT configuration One possible scenario would be to partition the image space by dividing it into sets of scan lines that could be distributed amongst the engines. Thus, each PERT operates independently on the set of scanlines allocated to it. Note that this places the multi-PERT machine in the *intelligent pixel* category. Memory contention for the frame buffer is avoided by partitioning the frame buffer, and providing each PERT with an independent portion of the frame buffer that stores pixel values associated with its scan line set.

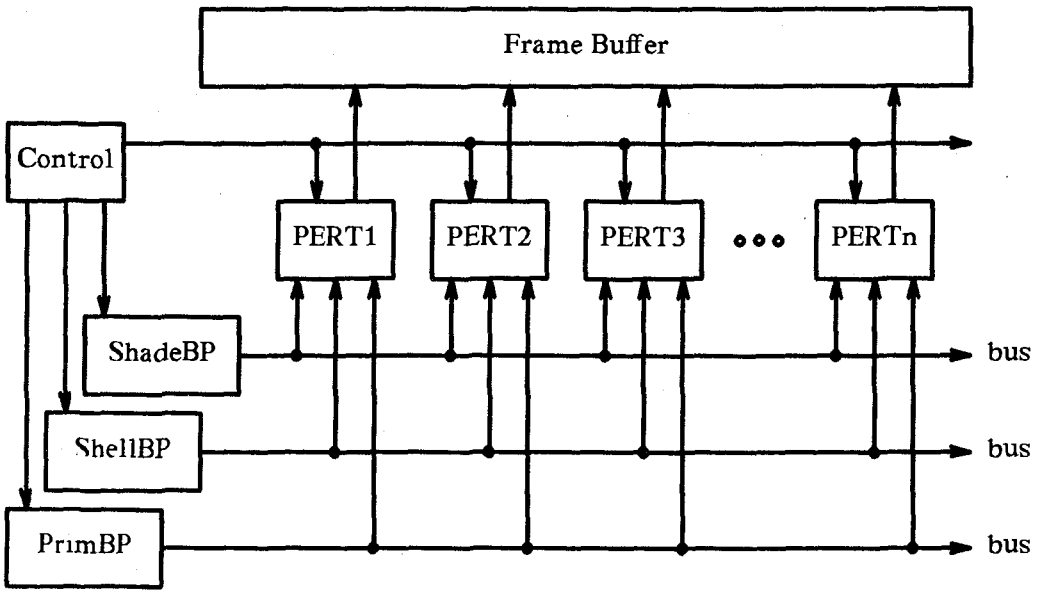

Figure 5-4: Multi-PERT configuration

5.2.3. Bus Interface Controller

The processors within a multi-PERT machine are not connected directly to the broadcast buses, but are connected through a device called the Bus Interface Controller (BIC) which reduces significantly the latency associated with accessing data from the broadcast bus.

Data on the broadcast bus is transmitted in form of *packets*. A *packet* has an identification number (ID) followed by a logical collection of data. The logical collection of data could be for instance, a set of shell data on the ShellBus, geometric description of a

set of primitives on the PrimBus, or reflective and refractive characteristics of a set of primitives on the ShadeBus. SJ16 makes requests for these packets by writing the desired IDs into a available ID-register within the BIC (see fig. 5-5). The BIC associatively matches each ID appearing on the broadcast bus with the contents of the ID-registers. If a match is found, two activities occur. First, a *hit* flag is raised, which is a signal to the broadcast processor to transmit the remaining data portion of the packet. Actually all the hit-flags of PERTs are inclusive-ORed, and the single output line sampled by the broadcast processor. The broadcast processor will only transmit the data portion of a packet if one or more PERTs raise their hit-flags. This drastically reduces the broadcast cycle time and hence latency time. Second, the data portion of the packet is copied on transmission, into a double buffer This way the BIC could be filling in packet information in one buffer, while SJ16 is reading the other. The double buffer is similar to the IMSU on the LINKS-1 machine, except that the individual buffers are FIFOs as opposed to being RAMs. The FIFOs permit SJ16 to read their contents in one cycle.

To summarize, the BIC offers 3 distinct advantages:

- It serves as an I/O processor for SJ16 by relieving it of data collection chores. Also, since it operates in parallel with SJ16, the overall processing time is reduced.
- It reduces latency since it has multiple ID-registers and looks for a match with any one ID contained in the registers.
- It reduces broadcast cycle time because of the hit-flag feedback to the broadcast processors that prevents data from being needlessly transmitted.

There are a host of other issues regarding broadcasting that are beyond the scope of this thesis. An exhaustive study of broadcasting in context of a multi-PERT organization is currently being undertaken as a master's thesis research project [GAUD85] by Severin

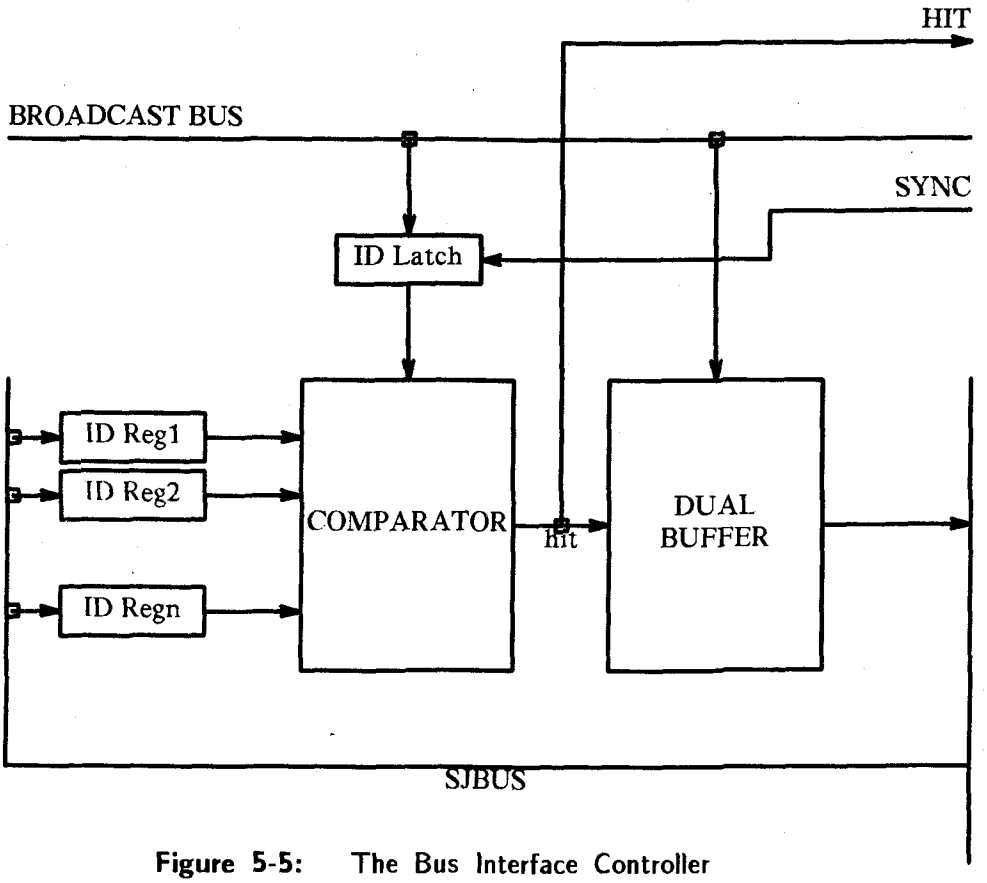

Figure 5-5: The Bus Interface Controller

Gaudet, who is simulating the broadcasting effects on PERT to study performance degradation because of access latency. Preliminary results seem to suggest minimal degradation, showing promise in the broadcasting technique for a multi-PERT organization.

Chapter 6

SIMULATION OF PERT

To evaluate the effectiveness of PERT, we decided to model it in software and run some benchmarks on it. As mentioned earlier, PERT derives its strength from two principal sources: a) the pipeline architecture, and b) the microcoded custom VLSI processor. Thus, to accurately simulate the ray tracing engine, we would have to provide simulation at two levels:

1. A higher-level simulation indicating the performance of the pipeline.
2. A lower-level microcode simulation to evaluate benefits accruing from microcoding the task algorithms.

Since in the process of developing our ray tracing package, we had specifically structured the software to reflect the tripartite distribution of work, the simulation at level 1 turned out to be straightforward. The simulator was easily added to the ray tracing software package.

The simulation at level 2 had two aspects to it. First, the SJ16 processor simulator was already available in the ASP environment, but we still had to build a software model for the floating point unit. Second, the task algorithms had to be microcoded, debugged, and timed in the ASP environment. Finally, one had to merge the results of simulation level 2 into the simulation level 1 to provide a complete simulation of PERT.

In the following sections we shall discuss in detail the two levels of simulation.

6.1. Level 1 Simulation

The ray tracing software package, which performed simulation at level 1, was written in C under UNIX, and was based on the algorithm described in Chapter 4. To accurately simulate the pipeline, the three software modules which emulate their hardware counterparts, were implemented as three independent processes under UNIX that communicated with each other through inter-processor communication *sockets*.

If we were to run the simulation *as is*, we would get the equivalent of PERT with a VAX 11/750 for each of its constituent processors. To simulate a real PERT with each of its constituent processors being constructed with SJ16, the timings collected from the various modules would have to be microcode timings. This is precisely the function of the simulation at level 2, which provides pre-computed microcode timings for each function. These timings are then inserted into the various modules.

6.2. Level 2 Simulation

In the following sections we will discuss the ASP to give a flavor of the environment used for microcoding, the FPU modeling by APL functions, and some issues about the microcode written.

6.2.1. Architecture Support Package

The Architecture Support Package [HOBS82, HOBS81b] is an APL workspace wherein SJ16 primitives have been modeled with APL functions. To illustrate this point consider the following example taken from [HOBS81b]. Assume we wish to add the contents of two registers. In natural APL this would be represented as:

```
R[X] <- R[X] + R[Y]
```

Unfortunately, the '+' in an ALU causes flags to be set as a side-effect. Such effects cannot normally be captured in APL. A simple solution is to define a diadic function PLUS which simulates the appropriate ALU action. The ASP therefore consists entirely of such user defined APL functions and global variables required by microprogrammers to drive SJ16.

Microcode is written as microAPL statements, a subset of APL. MicroAPL can be executed and as a side effect of microAPL execution a global variable in the workspace, called CLOCK, reflects the number of clock cycles required for the simulation run. The process of microcode development is best illustrated by a simple example. Assume we wanted to write a microalgorithm to evaluate the following fragment of code that computes ten numbers from the Fibonacci series starting from the third:

```
/* Compute Fibonacci numbers */

First  := 0;
Second := 1;
for i:= 10 downto 1
   begin
         Next   := First + Second;
         First  := Second;
         Second := Next;
   end
```

The corresponding microcode is shown in figure 6-1.

This simple example does illustrate some key features of the ASP environment. Recall that SJ16 has a built in counter for loop handling. The setup routine (fig. 6-1) is an APL function and *not* microAPL. It simply serves to initialize the CLOCK, call the function FIB1, and print the CLOCK value. The REG statement on line 5 is a compiler declarative that attaches the symbolic names FIRST, SECOND, and NEXT to registers R0, R1, and R2.

Figure 6-1: MicroAPL functions for Fibonacci series

```
      ∇ TEST∆FIB;FIRST;SECOND;NEXT
[1]       ⍝ NOTE THAT THIS IS NOT MICROAPL.
[2]       ⍝ IT IS USED AS A SETUP PROGRAM.
[3]       ⍝
[4]       STROINIT
[5]       REG FIRST←R0 ∆ SECOND←R1 ∆ NEXT←R3
[6]       FIB1
[7]   'CLOCK :',▼CLOCK
    ∇


      ∇ FIB1
[1]       ⍝ MICROAPL FOR COMPUTING FIBONACCI NUMBERS.
[2]       ⍝
[3]       R[FIRST]←COPY D '0'
[4]       R[SECOND]←COPY D '1'
[5]       COUNTER←NEGATE D '10'
[6]       COUNT
[7]   LOOP:R[NEXT]←R[FIRST] PLUS R[SECOND]
[8]       R[FIRST]←COPY R[SECOND]
[9]       R[SECOND]←COPY R[NEXT]
[10]      →LOOP IF~COUNT
    ∇


      ∇ FIB2
[1]       ⍝ IMPROVED MICROAPL FOR COMPUTING FIBONACCI NUMBERS.
[2]       ⍝
[3]       COUNTER←NEGATE D '10'
[4]       COUNT ∆ R[FIRST]←COPY D '0'
[5]       R[SECOND]←COPY D '1'
[6]   LOOP:R[NEXT]←R[FIRST] PLUS R[SECOND]
[7]       R[FIRST]←COPY R[SECOND]
[8]       →LOOP IF~COUNT ∆ R[SECOND]←COPY R[NEXT]
    ∇
FIB1 CLOCK: 44

FIB2 CLOCK: 33
```

Each microAPL statement, which gets translated by the compiler to one line of SJ16 microcode, consists of one or more microoperations. One is the norm, but under special circumstances it is possible to fit two or more microperations per microAPL statement. For example, it is possible to increment and test the counter in parallel with an ALU operation. We could thus, combine lines 9 and 10 of FIB1 and write one microAPL statement. We can also combine lines 6 and 4 of FIB1. The improved microAPL function FIB2 is shown in figure 6-1. The same figure also shows that it takes FIB1 44 clock cycles to execute, whereas FIB2 takes 33, a decrease of 10 clock cycles.

Note that FIB1 and FIB2 are directly APL executable since all SJ16 primitives PLUS, COUNT, etc. are actually APL functions that emulate the real primitives.

6.2.2. Modeling the FPU

As mentioned earlier, the selection criteria for the floating point coprocessor was mainly the throughput speed. The Weitek [WEIT85b] WTL 1164/65 low-latency chip set, capable of performing both 32 and 64 bit floating point arithmetic, is possibly the fastest coprocessor on the market today. The WTL 1164 Floating Point Multiplier can do a 32 bit multiply in 360 nsec. The WTL 1165 Floating Point ALU can perform a 32 bit add/subtract/convert/compare in 360 nsec. Besides, the WTL 1165 ALU is also capable of performing a 32 bit floating point divide in 1.86 μsec.

Before we can model in software the Floating Point Unit, we need to conceptualize in hardware the internal organization — the control and the datapaths of the floating point subsystem. The proposed FPU organization is shown in figure 6-2. It consists of two subunits, the ALU and the multiplier MUL, which can operate concurrently.

The A and B input registers, which are 32 bits wide, buffer data coming in from the 16

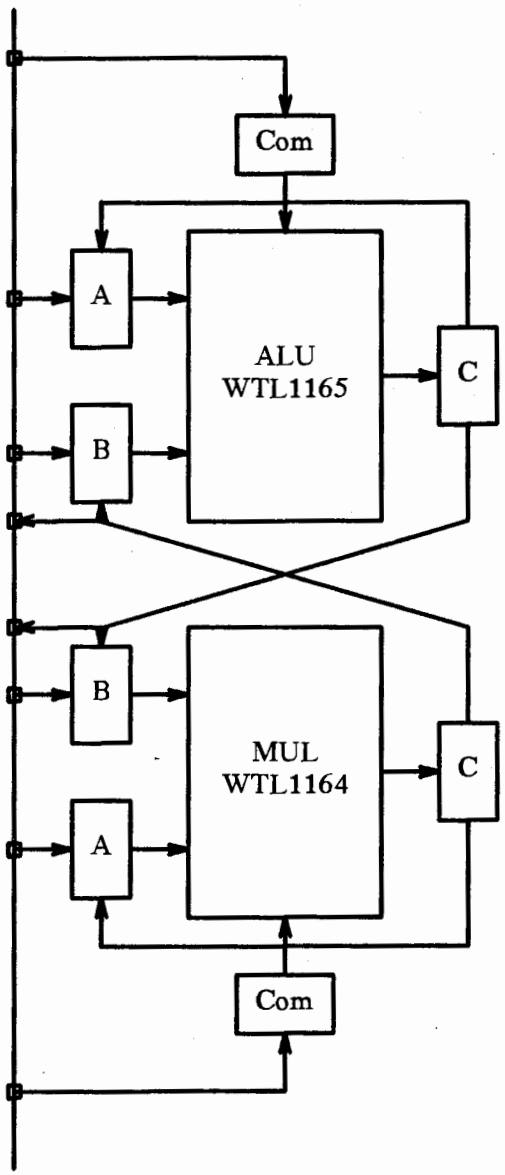

Figure 6-2: Organization of the Floating Point Unit

bit SJBUS, since the FPU operates at a speed much higher than SJ16 (60 nsec. clock vs. an estimated 200 nsec.). The C register buffers a 32 bit result which can be read onto SJBUS, 16 bits at a time. The two 8 bit command registers hold commands for the subunits, and can be loaded in parallel, in one SJ16 clock cycle. Loading the command register also implicitly starts an FPU operation.

In performing successive floating point operations, it is sometimes useful to direct the output of one subunit into either one of its own input registers or into the input register of the other subunit. This is done within the same time slot as that needed to complete a floating point operation. In absence of such data paths, an extra data transfer cycle would be needed to transfer data from the output register into an input register. There could be 8 possible data paths — 4 from output of ALU to input registers and 4 from output of MUL to input registers. Complete realization of all data paths would entail a lot of multiplexor circuitry. As a compromise we decided to use only 4 data paths as shown in figure 6-2. Note that this is an acceptable compromise since for one, multiplication is commutative, and hence there need only be one path from C_{ALU} to MUL inputs, and secondly the path from C_{MUL} to A_{ALU} was rarely used. In the rare event of a C_{MUL} to A_{ALU} transfer, an extra data transfer cycle must be used. In general, this model was found to be quite tractable within the framework of our arithmetic expressions.

The control of the data transfer is encoded in the command words broadcasted to the subunits. The 2 most significant bits of the ALU command word are to be interpreted as follows:

00	no transfer
01	$C_{ALU} \rightarrow A_{ALU}$
10	$C_{ALU} \rightarrow B_{MUL}$
11	$C_{ALU} \rightarrow A_{ALU}$ and $C_{ALU} \rightarrow B_{MUL}$

The remaining 6 bits are the function codes. The encoding for the MUL is similar and can be obtained by interchanging the register subscripts show above.

There are actually 2 more registers, not shown in figure 6-2, which are status registers. The status register holds the exception or conditions codes if any and for a compare operation on the ALU, status codes 0,1,2 represent $=$, $<$, and $>$ respectively. To provide more flexibility we provide a extended status register that derives extended status codes of $\neq$, $\leqslant$, and $\geqslant$ using simple combinational hardware. After a comparison function in the ALU, the extended status register is read into the SJ16 status register where general purpose bit-testing can be performed to provide the entire gamut of comparison operators.

Based on the proposed hardware FPU model, APL functions to emulate the FPU were written.

6.2.3. Microcoding the Task Algorithms

At first glance, attempting to microcode 3500 lines of C code appears to be a gargantuan task. It turns out, however, for the ray tracing algorithm, a major percentage of the execution time is spent in the *inner loops* of the algorithm that comprises relatively few lines of code. Thus an early decision was made regarding microding — we would microcode only the inner loop code, and from the timings obtained, extrapolate the timings for the entire task algorithms. To ensure reasonable simulation, we made sure that the modules microcoded represented at least 80% of the total execution time. This was determined by the UNIX profiler *gprof*. Table 6-1 gives an idea as to how much of the C code was actually microcoded for each of the 3 task algorithms. It represents the percentage of the total execution time taken up by functions that were actually microcoded.

ShellProc	81%
PrimProc	94%
ShadeProc	95%

Table 6-1: Percentage of total execution time for microcoded functions

6.3. Merging Simulation Results

Once microcode timings were available for the three tasks, this information was embedded into the C code. Thus, when a C function is called, it increments a global variable INT_CLOCK, by and amount equal to the number of clock cycles taken by an equivalent function in microcode. We could have avoided the awkwardness of having to run microcode simulations on one machine (IBM 3033 running the Michigan Terminal System), and then transferring the results to the other simulation (running under UNIX on a VAX 11/750), by writing our ray tracing package in APL. But then APL is far too *slow* in execution to be used realistically as a rendering tool, which we hoped our software package would eventually be used for. Besides, under UNIX we had access to a whole set of software tools, which we used extensively, but which were unavailable under MTS.

Chapter 7

RESULTS AND CONCLUSION

7.1. Results

Results are presented in three parts. First, the timings of a microcoded (SJ16) function versus the timings of the same function implemented in C on a VAX 750, are shown. Second, a table showing the performance of the pipeline: the processing and the idle time for each processor, and pipeline efficiency, is presented. Third, the overall execution time on a VAX 11/750 and the simulated execution time on PERT are presented. All timings are based on the assumption that SJ16, a 16 bit processor, operates at a clock speed of 5 Mhz.

7.1.1. Microcode Timings

Table 7-1 shows microcode timings for CheckSphereIntersection, a function that computes sphere-ray intersections. These timings are compared with the timings of same function coded in C, on our 2 VAXs: one with floating point hardware and one without.

All floating arithmetic in C is carried out in double-precision; whenever a *float* appears in an expression it is lengthened to *double* by zero-padding its fraction [KERN78]. The microcode timings, on the other hand, are based on single-precision arithmetic. A straightforward comparison, therefore, would be unfair. To rectify this, we took the assembly output of CheckSphereIntersection and changed it so that all floating point operations were done in single-precision. Incidentally, double-precision floating point

operations on the VAX take approximately 25% more time than their single-precision counterparts [HOBS84], a fact that needs to be considered while comparing overall VAX time with PERT.

VAX 750 with FPU	VAX 750	SJ16 with FPU
704 μs	2193 μs	54.6 μs

Table 7-1: Timings for the function CheckSphereIntersection

7.1.2. Pipeline Timings

Table 7-2 illustrates pipeline performance. The idle time for each processor is compared with the processing time. A utilization factor (η) is shown for each scene where η is defined as:

$$\eta = \frac{(T_p)}{(N \times T_\pi)} \times 100$$

- T_p is the total processing time (sum of all processing times).
- T_π is the time taken by the pipeline.
- N is the number of processors (3) in the pipeline.

Notice that the shader has a comparatively high idle time. A more sophisticated shader — for instance, a full implementation of Hall's shading model — would make the pipeline more balanced, or a slower cheaper processor could be used.

The scenes used for testing were generated to test different combinations of tree order and the number of primitives per leaf shell; the 2 in scene 208 indicates a binary tree and

the 08 indicates that there were 8 primitives per leaf shell. There were a total of 512 spheres in each of the three scenes. The timings shown are for a raster size of 64 x 48 pixels.

7.1.3. VAX 11/750 versus PERT

Table 7-3 shows the overall rendering times for 5 scenes on a VAX 750 (with FPU) and PERT. In addition to the scenes discussed in the previous section, timings for scenes 23 and 45 which are moderately complex scenes (1093 and 2754 primitives) are also shown. Scene 23 is shown in figure 1-4 and scene 45 in figure 7-1. The bubble bodies in scenes 23 and 45 were generated using the SFU kinematic simulation system [CALV82].

For the different types of scenes, PERT is on average, 24 times faster than VAX. Recall from earlier sections that an adjustment up to 25% may need to be made for VAX timings because of the double-precision overhead. Overall timing reduction, however, would be less than 25% because the actual floating-point computation time forms only a fraction of the total execution time. Even with a 25% decrease in VAX timings, PERT is approximately 18 times faster.

7.2. Discussion

7.2.1. Processor improvement

All timings generated so far were based on a 5Mhz clock for SJ16. This was done because our current version of SJ16 runs at roughly 5Mhz. With plunging feature sizes in current CMOS technology, it appears possible to fabricate SJ16 running at a clock speed of 8 Mhz by going to a 3 micron process. It is even possible to fabricate SJ32, a 32-bit version of SJ16. The effect of these changes on the running time of the function

SCENE 208	Processing time	Wait time
ShellProcessor	2.88	0.58
PrimProcessor	2.55	0.91
ShadeProcessor	1.40	2.07
$\eta = 65.77\%$		
SCENE 408	Processing time	Wait time
ShellProcessor	3.22	0.39
PrimProcessor	2.55	1.06
ShadeProcessor	1.40	2.22
$\eta = 66.15\%$		
SCENE 808	Processing time	Wait time
ShellProcessor	4.23	0.25
PrimProcessor	2.58	1.90
ShadeProcessor	1.41	3.07
$\eta = 61.18\%$		

Table 7-2: Pipeline processing and wait times for 3 sample scenes.

SCENE	VAX 750 (sec)	PERT (sec)
Scene 208	092.10	3.46
Scene 408	092.91	3.62
Scene 808	108.50	4.48
Scene 23	101.02	4.42
Scene 45	138.03	6.73

Table 7-3: Total times taken by VAX and PERT

CheckSphereIntersection is shown in table 7-4. SJ32 performance at 5 Mhz is almost the same as SJ16 at 8 Mhz, but SJ32 would consume a substantially larger chip area and would need a more expensive memory interface. The saving in chip area for SJ16 could be used for the BIC, thus reducing chip count.

The exact overall performance improvement in the running time is difficult to predict. Since FPU execution time does not change with the clock rate or bus width, the improvement depends upon the fraction of the time spent by the tasks in data transfer. The best case running time would be obtained if there were no floating point operations. For SJ16, the percentage decrease in running time by going from a 5Mhz to 8Mhz processor would be:

$$\frac{(200-125)}{200} = 37.5\%$$

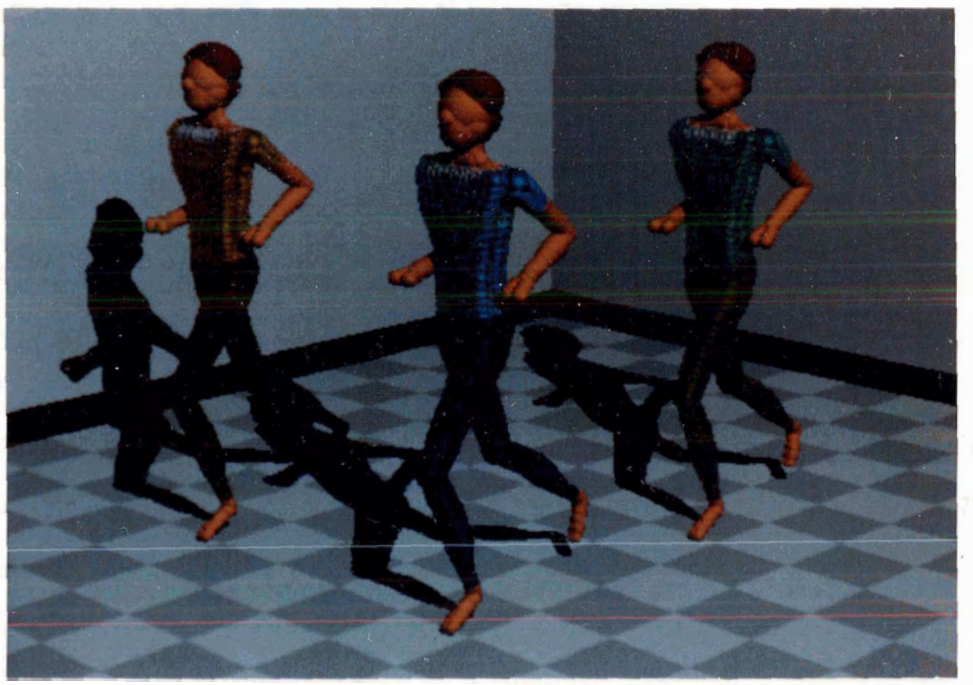

Figure 7-1: Scene 45 used in VAX-PERT timing comparisons

SJ16 5Mhz	SJ16 8Mhz	SJ32 5Mhz	SJ32 8Mhz
54.60μs	39.75μs	40.40μs	30.88μs

Table 7-4: Running times with improvement in processors

To compute a lower bound for performance improvement let us examine what a floating point operation entails. First, we must transfer at least 2 data words to the FPU. The FPU then takes constant time to execute the operation. If t_c is the clock time, and t_{fpu} the FPU execution time. Then the total time required to perform a complete floating point operation is:

$$t_f = 2t_c + t_{fpu}$$

If the clock time is changed to t_c' then the new time is:

$$t_f' = 2t_c' + t_{fpu}$$

Therefore the percentage improvement in time is:

$$\frac{t_f - t_f'}{t_f} = 18.75\%$$

where t_c is 200ns, t_c' is 125ns, and t_{fpu} is 400ns. The value of t_{fpu} actually depends upon the floating point operation. For add, subtract, and multiply it is 400ns, for a divide it is 1.86μs. Since divide is infrequently used, t_{fpu} was chosen to be 400ns. For SJ16 running at 8Mhz, therefore, the improvement in running time would be at least 18.75% and would not exceed 37.5%.

7.2.2. Multi PERT performance

Figure 7-2 shows the overall improvement in performacne with the increase in the number of PERTs used to construct a multi-PERT organization. The results are based on scene 45 (see figure 7-1). The Y-axis represent the ratio of the rendering time on VAX and the rendering time on the multi-PERT machine. For more details the reader is referred to [GAUD85].

7.2.3. Host-PERT interaction

To produce a complete working rendering system, PERT would have to be connected to a host machine, typically a graphics workstation The host maintains all the data sets on disk, and uses the PERT as a peripheral. The host initiates a frame rendering by loading PERT with the three data sets. The host is then free to start pre-processing the next frame of data while PERT works on the last one, giving rise to additional parallelism.

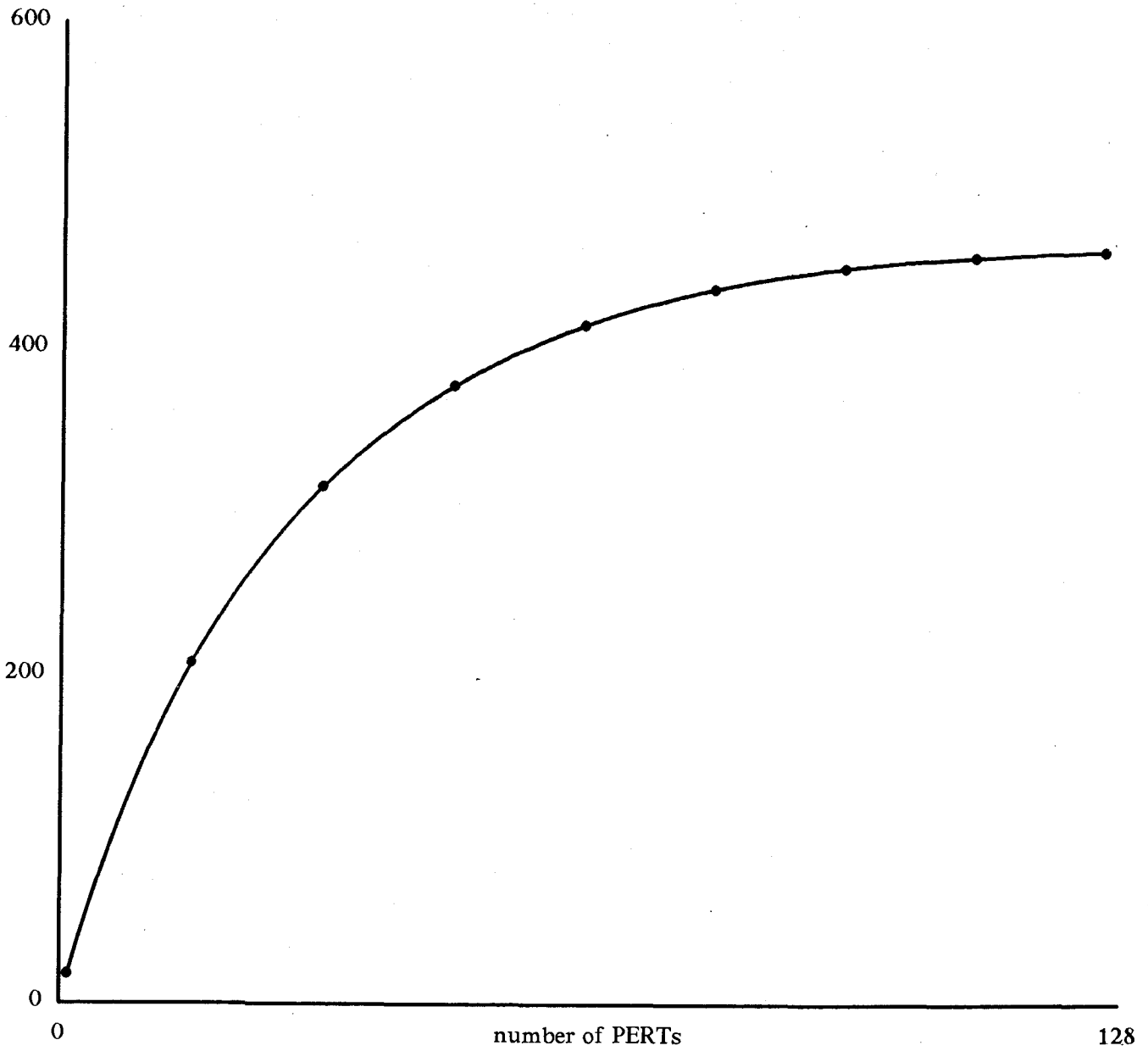

Figure 7-2: Number of PERTs vs. Performance

7.2.4. Advantages & Disadvantages

The biggest advantage of PERT is that it is modular. Many PERTs can easily be connected to form a multi-PERT machine for ray tracing. Even a single-PERT configuration is at least 20 times faster than the VAX. Besides, unlike Ullner's ray tracing peripheral or ray tracing pipeline, there is no limitation on the type of surface that can be used for modeling. New surfaces can be added by merely changing the microcode of the PrimProcessor to perform intersections with new surfaces. Using microprogrammable processors allows for a more flexible PERT which can be tailored to applications instead of being a fixed hardware solution. The potential for downloading microcode means that one could extend the ShellProcessor to handle other bounding volume shapes; add to the PrimProcessor functions for other geometric and procedural objects; and extend the ShadeProcessor to handle better shading models and techniques such as texture mapping.

On the other hand, a drawback of this architecture is that is depends very much on the way the user chooses to hierarchically structure his data. If the user chooses a hierarchical tree that causes imbalance amongst the processors — a pathological case being no hierarchy at all — then serious degradation may occur. Of course, such a degradation would occur even on a uni-processor system.

7.2.5. Extensions

One question remains to be answered: Given a fixed number of primitives what is the best possible way to hierarchically structure data to achieve minimum rendering time on PERT? Is it better to have binary trees, quad trees, octrees? What is the best number of primitives to have per leaf-shell? These questions cannot be fully answered until further investigation is done. Table 7-5 shows the total rendering time for different combinations of tree order and number of primitives/leaf-shell on a scene consisting of 512 spheres. The

	Number of Primitives/Shell					
	2	4	8	12	16	20
Binary tree	3.36	3.18	3.46	4.19	5.25	6.48
Quad tree	3.69	3.42	3.62	4.27	5.28	6.50
Oct tree	4.82	4.33	4.48	5.03	5.69	6.64

Table 7-5: Timings for different combinations of tree order and prim/shell

least rendering time (3 18 secs.) was taken by a binary tree, with 4 primitives/shell. Although the table indicates a clear advantage in lower order trees and lower number of primitives/leaf-shell, the results cannot be generalized and applied to scenes with a mix of different types of primitives.

If we can identify a clear strategy for structuring data, or maybe even reduce it to a set of heuristic rules, it should be possible to write a front-end program to automatically cluster data into a hierarchical tree. Such a program would then free the naive user from deciding how to organize his data, or use its expertise to aid a more experienced user in building the hierarchy.

7.3. Conclusion

In this thesis a 3-processor pipelined engine model, PERT, has been presented which executes a ray tracing algorithm that has been subdivided into 3 tasks. The entire scene data is also partitioned into 3 data sets; one for each task. PERT is highly modular — many such individual PERTs can be connected in parallel with the scene data being globally distributed on 3 buses by means of *broadcasting*. Besides, microcode can be

downloaded into PERT, providing flexiblity in handling a wide variety of bounding volumes, modeling surfaces, and shading models.

Simulation results show that a single PERT, in itself performs about 20 times faster than a VAX 11/750 with floating point hardware. Multiprocessor simulation results [GAUD85] indicate performance improvements greater than two orders of magnitude with 8 PERTs in a multi-PERT configuration using sample scenes of moderate complexity.

Although we have partly addressed the issue of creating optimal data trees which keep the constituent processors in PERT balanced, further work needs to be done in this area, especially when considering scenes which have primitives with widely varying ray intersection costs.

References

[APPE68] Appel, A.
Some techniques for shading machine renderings of solids.
In *AFIPS Spring Joint Conference*, pages 37-45. AFIPS, 1968.

[BLIN77] Blinn, J.F.
Models of light reflection for computer synthesized pictures.
In *Siggraph'77 Conference Proceedings*, pages 192-198. ACM, San Jose, California, 1977.

[CALV82] Calvert, T.W., Chapman, J., and Patla, A.
Aspects of the Kinematic Simulation of Human Movement.
IEEE Computer Graphics and Applications 2(9):41-49, November, 1982.

[CLAR76] Clark, J.H.
Hierarchical geometric models for visible surface algorithms.
Communications ACM 19(10):547-554, October, 1976.

[CLEA83] Cleary, J.G., Wyvill, B., Birtwistle, G. M., and Vatti, R.
Multiprocessor ray tracing.
Technical Report 83/128/17, University of Calgary, October, 1983.

[COOK82] Cook, R.L., and Torrance, K.E.
A reflection model for computer graphics.
ACM Transactions on Graphics 1(1):7-24, January, 1982.

[DIPP84] Dippe, M., and Swensen, J.
An adaptive subdivision algorithm and parallel architecture for realistic image synthesis.
In ACM (editor), *SIGGRAPH'84 Conference Proceedings*, pages 149-157. ACM, New York, 1984.

[FLYN66] Flynn, M.J.
Very high-speed computing systems.
In *Proceedings of the IEEE*, pages 1901-1909. 1966.

[GAUD85] Gaudet, S.
A parallel architecture for ray tracing.
Master's thesis, Simon Fraser University, May, 1985.

[GLAS84] Glassner, A.S.
Space subdivision for fast ray tracing.
IEEE Computer Graphics and Applications 4(10):15-22, October, 1984.

[GOLD71] Goldstein, R.A., and Nagel, R.
3-D visual simulation.
In *SIMULATION*, pages 25-31. January, 1971.

[HALL83] Hall, R.A., and Greenberg, D.P.
A testbed for realistic image synthesis.
IEEE Computer Graphics and Applications 3(10):10-20, November, 1983.

[HOBS81a] Hobson, Richard.
Structured Machine Design: An Ongoing Experiment.
In *Proceedings of the 8th Symposium on Computer Architecture*, pages 37-55. SIGARCH, Minneapolis, May, 1981.

[HOBS81b] Hobson, Richard, Hannon P., and Thornburg J.
High-level Microprogamming with APL syntax.
Technical Report TR 81-2, Simon Fraser University, 1981.

[HOBS82] Hobson, Richard.
SAMjr Microprogamming guide, Version 2.1.
1983.

[HOBS84] Hobson, R., Gudaitis, J., and Thornburg, J.
A New Machine Model for High-Level Language Interpretation.
Technical Report CMPT TR 84-18, Simon Fraser University, 1984.

[KAY79] Kay, D.S.
Transparency, refraction, and ray-tracing for computer synthesized images.
Master's thesis, Cornell University, January, 1979.

[KERN78] Kernighan, B.W., and Ritchie, D.M.
The C Programming Language.
Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1978.

[NISH81] Nishimura, H., Ohno, H., Kawata, T., Shirakawa, I., and Omura, K.
LINKS-1: A parallel pipelined multimicrocomputer system for image creation.
In ACM (editor), *Proceedings of the 10th Symposium on Computer Architecture*, pages 387-394. ACM, New York, 1981.

[PHON73] Phong B-T.
Illumination model for computer generated pictures.
Communications ACM 18(6), June, 1975.

[RUBI80] Rubin, S.M., and Whitted, T.
A 3-dimensional representation for fast rendering of complex scenes.
In ACM (editor), *SIGGRAPH'80 Conference Proceedings*, pages 110-116. ACM, New York, 1980.

[ULLN83] Ullner, M.K.
Parallel machines for computer graphics.
PhD thesis, California Institute of Technology, 1983.

[WEGH84] Weghorst, H., Hooper, G., and Greenberg, D.P.
Improved computational methods for ray tracing.
ACM Transactions on Graphics 3(1):52-69, January, 1984.

[WEIT85a] Weitek Solids Modeling Engine.
Weitek Corporation Product Literature. 1985.

[WEIT85b] WTL1164/1165 Low-Latency 64-bit IEEE Floating Point Multiplier/ALU.
Weitek Corporation Product Literature. 1983.

[WHIT80] Whitted, T.
An improved illumination model for shaded display.
Communications ACM 23(6):343-349, June, 1980.